NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

MULTIPLE ROBOTS LOCALIZATION VIA DATA SHARING

by

Cheng Leong Ng

September 2015

Thesis Advisor:	Oleg Yakimenko
Co-Advisor:	Roberto Cristi

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE MULTIPLE ROBOTS LOCALIZATION VIA DATA SHARING			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ng, Cheng Leong			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**
13. ABSTRACT (maximum 200 words) This thesis applies a systems engineering approach to identify the critical issues in using a robot localization technique for a swarm of unmanned systems operating in an urban environment. It starts by presenting a concept of operations requiring data sharing between multiple robots operating in a confined environment, and proceeds with the development of a localization technique based on observing the relative position of neighbor vehicles and then sharing this information with them. The centroids of the measured positions are fed into a Kalman filter as the measurement inputs. The Kalman filter merges measurement data with a predicted state from a simple kinematic model. A simulation developed in Python is used to compare the performance of developed data-sharing localization technique with the individual robot odometry. The simulation results show a significant improvement of robot localization precision while the simple odometry technique results with continuing growth of the estimation error.			
14. SUBJECT TERMS robotics, robot localization, collaborative robotics, urban environment, Kalman filter, data sharing			**15. NUMBER OF PAGES** 105
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

MULTIPLE ROBOTS LOCALIZATION VIA DATA SHARING

Cheng Leong Ng
Civilian, Singapore Technologies Dynamics, Singapore
B.Eng (Electrical), National University of Singapore, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Approved by: Oleg Yakimenko
Thesis Advisor

Roberto Cristi
Co-Advisor

Ronald Giachetti
Chair, Department of Systems Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis applies a systems engineering approach to identify the critical issues in using a robot localization technique for a swarm of unmanned systems operating in an urban environment. It starts by presenting a concept of operations requiring data sharing between multiple robots operating in a confined environment, and proceeds with the development of a localization technique based on observing the relative position of neighbor vehicles and then sharing this information with them. The centroids of the measured positions are fed into a Kalman filter as the measurement inputs. The Kalman filter merges measurement data with a predicted state from a simple kinematic model. A simulation developed in Python is used to compare the performance of developed data-sharing localization technique with the individual robot odometry. The simulation results show a significant improvement of robot localization precision while the simple odometry technique results with continuing growth of the estimation error.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis uses a systems engineering approach to identify the critical factors necessary for the robot localization technique to satisfy the needs of an autonomous unmanned system in an urban operation. This is done by analyzing the problem definition of robot localization in detail. With the boundaries of the problem identified and the stakeholders analyzed, a set of requirements for the system is defined. By performing functional analysis, the vital functions of robot localization can be determined. To this end, the concept of operation of a data-sharing robot localization was designed.

The data-sharing robot localization technique designed involves a robot measuring the position of its peers and sharing out the information. The centroid of the measured position data are fed into a Kalman filter as the measurement inputs. The Kalman filter combines the measurement data with a predicted state from a kinematics model.

A simulation developed in Python is then used to compare the performance of the proposed data-sharing robot localization with the individual robot odometry. The simulation results of the technique show promising improvements in robot localization when the odometry errors are significantly larger than the measurement errors.

From experiments done using the simulation, it was seen that by increasing the number of robots in the group, the performance of the data-sharing robot localization improved. The improvement to robot localization by data sharing is greater when the odometry errors are bigger. However, it is noted that the measurements done by the observing robots are erroneous as well. Hence, when odometry errors are small, data sharing will actually make the performance of the robot localization worse.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Professor Oleg Yakimenko for sharing his valuable advice and expertise during the course of this journey. He has always been approachable and supportive of the ideas that I have.

I would also like to thank Professor Roberto Cristi for his advice, especially in the area of designing and implementing Kalman filters.

Finally, I would like to express my gratitude to my family who is with me in Monterey for their support during the course of my studies at Naval Postgraduate School.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Imagine walking around a room with your eyes closed. There are obstacles like tables and chairs in the room. You will probably find it rather difficult to find your way even if you know exactly what route you need to take. It is only because we have eyes to tell us where we are in every movement that we are able to easily navigate in the world. Our eyes are constantly correcting the errors we make in our movement. The inaccuracies of our movement become more apparent when we try to walk in a straight line with our eyes closed.

When moving around the world, a robot face the same problem. It needs sensors in the same way a human needs eyes to detect the inaccuracies in its movements. If a robot move around without feedback, it will get lost because of imperfections in the mechanisms enabling its movement as well as the changes that take place in a dynamic environment.

B. MOBILE ROBOTS

The key motivation driving the research into a mobile robot is its potential to replace the need for humans in the following three types of jobs:

- dirty
- dangerous
- dull

Jobs such as firefighting, search and rescue missions, toxic waste cleanup, nuclear power plant decommissioning, security, and surveillance and reconnaissance tasks all contain possible risks of human casualties. Some of these jobs involving long hours are simply boring. In all of these jobs, there is a desire to reduce the direct involvement of humans. This can be done by replacing humans with robots.

With increased urbanization occurring across the globe, it can be expected that the urban operating environment is going to be the most important environment that authorities should pay attention to. In such an environment, when a disaster occurs, whether natural or man-made, there will be a need for immediate search and rescue efforts. Such operations are often dangerous, labor-intensive and information-scarce. This is exactly the type of operation for which an autonomous robot is best suited.

C. AUTONOMOUS ROBOT

The autonomy of a robot depends on the extent to which the robot needs prior knowledge of its environment in order to complete its tasks. Autonomy can be classified into three classes: non-autonomous, semi-autonomous, and fully autonomous (Negenborn 2003).

(1) Non-autonomous

A non-autonomous robot is completely controlled by humans remotely. The only intelligence possessed by the robot is the ability to interpret the commands sent by the operator control unit.

(2) Semi-autonomous

A semi-autonomous robot can either be controlled by humans or navigate by itself. This is useful in situations where human commands are delayed. The form of control can be either actual steering or in the form of a map given to the robot by the human controller.

(3) Fully autonomous

A fully autonomous robot is capable of fully steering itself. There is no requirement for human interaction in order for the robot to complete its tasks. The robot is capable of intelligent movement and action without any guidance externally to control it.

The need for autonomy is largely dependent on the situation. For example, in a factory setting, a robot manipulator should be non-autonomous, as the operator would not want the robot to perform its own action. A predictable action is preferred, and it would be less expensive and more reliable.

However, when the robot is performing tasks in which the operator has limited or no information of the exact set of actions to take, a semi-autonomous or fully autonomous robot should be used.

D. ROBOT NAVIGATION

Robot navigation is the task of an autonomous robot moving from one point to another. The ability to navigate is important for any mobile entity. Avoiding dangerous situations like unsafe conditions or collisions is important; however, if a robot's goal is to get to a specific spot, it would still have to find that place. Therefore, this problem can generally be crafted with three questions:

- *Where am I?* For a robot to make any decisions to move, it first has to determine where it is in its own frame of reference. This is usually called robot localization (Negenborn 2003).
- *Where am I going?* For a robot to complete any task, it has to know where to go to. This is the goal of the robot, and it is usually called goal recognition (Negenborn 2003).
- *How do I get there?* When a robot knows where it needs to go to and it knows its own position, the robot needs to find a way to get to its goal. This task is usually called path planning (Negenborn 2003).

E. ROBOT LOCALIZATION

This thesis focuses on the first question in robot navigation, which is the robot localization problem. Some authors consider the robot localization problem to be the most fundamental problem in providing a robot with truly autonomous capabilities (Cox 1990). The problem of localization is that of learning the spatial model of the robot's surrounding environment.

In order to sense its surroundings, the robot has to possess sensors to perceive the world. Sensors that are commonly used include sonar, laser, infrared range finders, cameras, radar, tactile sensors, compasses, and the Global Positioning System (GPS).

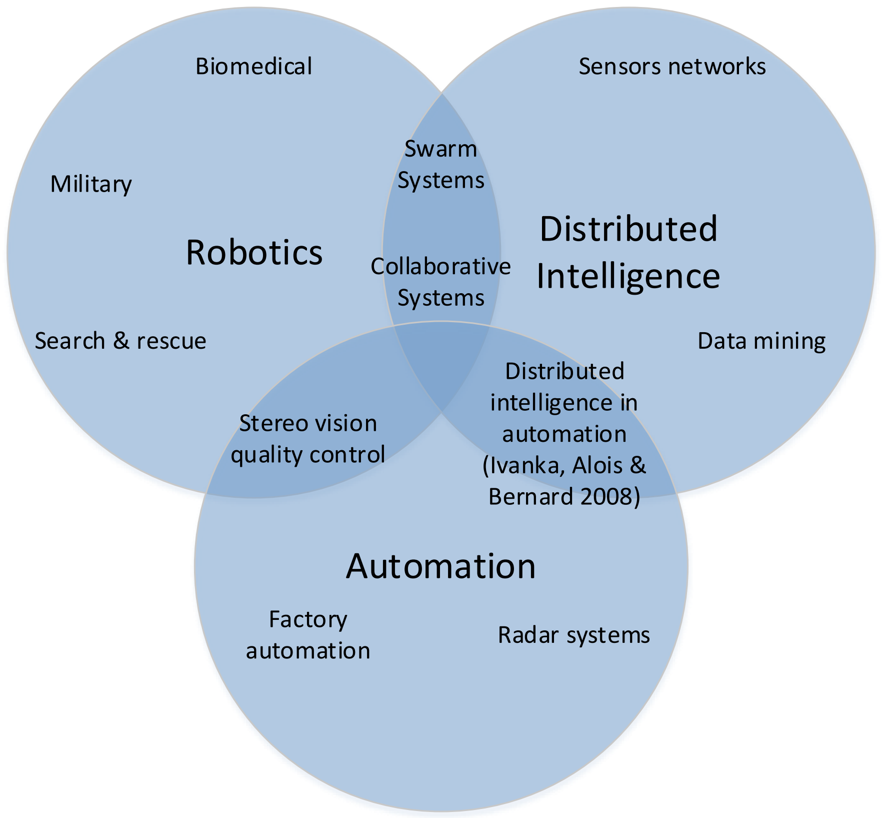
Robot localization can be loosely grouped into two categories, external robot localization and internal robot localization. In external robot localization, there is a need for additional supporting infrastructure such as external sensors or external localization reference. External sensors can include cameras or infrared sensors, and external localization reference can include radio beacons, LED emitters, ceiling projection, or GPS. External robot localization techniques have the advantage of providing "ground truth;" hence, the perceived position of the robot does not drift over time. However, the disadvantage of external robot localization is that existing infrastructure has to be in place in order for it to work. For the case of GPS, it would not work indoors, and in times of war, adversaries can jam the signal and create a GPS denied area.

Internal robot localization techniques such as simultaneous localization and mapping (SLAM)-based approaches are fundamental for an autonomous robot (Krajnik et al. 2013). They allow a robot to be truly mobile; however, the tradeoff is that these techniques are usually computationally intensive and the localization data are susceptible to drift. As such, the movement of the robot is often slow.

F. DISTRIBUTED INTELLIGENCE

Distributed intelligence has the objective to create systems that can collaborate in a way that they have the same level of performance and efficiency as human teams. These systems can be people, robots, computers, software agents, sensors or animals (Parker 2008). Such systems can be very helpful in addressing the many challenges we face today in urban search and rescue, computer security, military operations, logistics, and many other activities. This is a topic of interest to this thesis since many applications of distributed intelligence can be leveraged in the areas of robotics and automation, as depicted in Figure 1.

Figure 1. Applications in Distributed Intelligence, Robotics and Automation



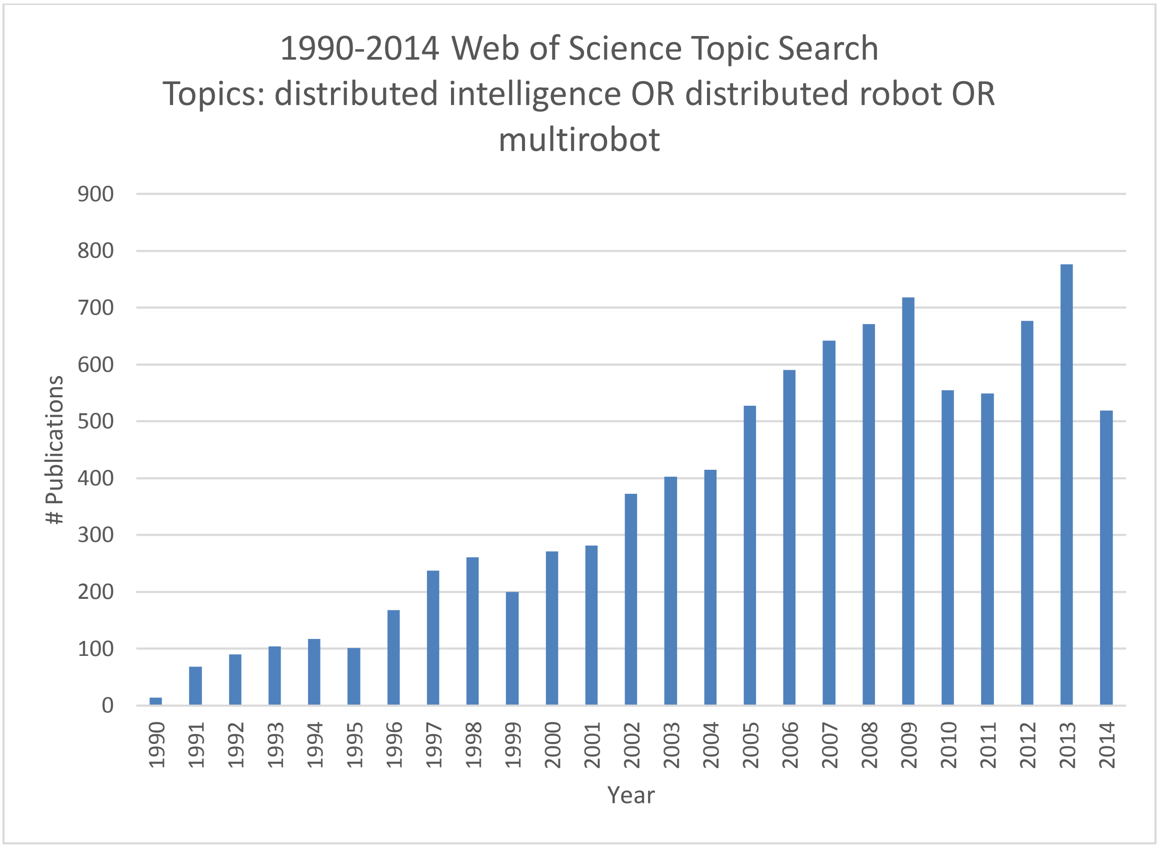
When considering the various problems to be solved by robots, one approach is to design a single robot capable of handling all the tasks necessary to solve any problem the robot may encounter. However, this robot would have to be designed to have all the capabilities needed in order to complete the task on its own. Very often, for small-scale jobs, this is sufficient and feasible.

However, when we look at the real world, many solutions to these problems involve teams of humans. Instead of having one human performing all the tasks alone, multiple humans, each with specialized skills complementing each other, work to create the solution. Hence, there is a motivation to think in terms of distributed systems.

As such, distributed intelligence refers to a group of entities working as a system to solve problems, reason, and plan (Parker 2008). In this case, an entity is defined as any type of intelligent system or process. In these systems, different entities usually specialize in different tasks or in certain aspects of a task.

It can be seen from a search in the Web of Science that topics related to distributed intelligence have actively been researched in the last couple of years. As such, it can be expected that there will be many applications of distributed intelligence, specifically in robotics, being developed.

Figure 2. Web of Science Data Showing Number of Publications Related to Distributed Intelligence from 1990 to 2014



G. PROBLEM FORMULATION AND ORGANIZATION OF THE THESIS

As discussed earlier, robot localization is one of the most fundamental problems with an autonomous robot. In addition, the development of multi-robot systems is expected to increase greatly across many applications. For these reasons, I would like to use this thesis to help improve the area of robot localization.

This thesis explores the idea of multiple robots assisting each other in performing robot localization with the hope of improving individual and overall accuracy as well as reducing the computational load of each individual robot.

This thesis is organized as follows. First, the Kalman Filters is discussed in order to give the reader an idea how measurements and predictions of states can be merged. Next a system engineering approach to analyzing robot localization is done to identify the critical factors. With the analysis done, a concept of operation and robot localization technique is developed. Next the thesis elaborates on the simulation model developed to examine the performance of the robot localization technique developed. Thereafter, the research scenario used to perform the experiment in the simulation is explained and the results of the experiments are analyzed. Lastly the conclusions and the recommendations of the thesis is discussed.

H. BENEFITS OF STUDY

This study explores methods of improving localization data. Specifically, it helps provide more accurate data, reduces drift in localization data, and reduces the need for more computationally intensive methodology in localization. In addition, the study allows parameters that impact the performance of robot localization to be identified. It will also enable the examination of the relationship of the parameters to the performance.

THIS PAGE INTENTIONALLY LEFT BLANK

II. KALMAN FILTERS

A. BACKGROUND

Generally speaking, the Kalman filter (Welch and Bishop 2006) is a recursive linear estimator that repeatedly generates an estimate for the state of a noisy linear dynamics system by minimizing the mean of the squared error.

The state of a system in this case would be a vector *x* consisting of a number of variables that describe some properties of interest in a system. For example, the vector *x* could consist of the position and velocity of a robot.

In the above case, the states are noisy and not easily observable. Hence, it makes the job of estimating the state a difficult task. In order to estimate the state, the Kalman filter requires a measurement of the system. The measurement has to be linearly related to the state. It is expected that the measurements are affected by errors. The Kalman filter estimator is statistically optimal with reference to any quadratic function of error in estimation if the errors are white noise (Mohinder and Andrews 2014).

With data of the initial conditions, the Kalman filter estimates the state by using all available information of the system and the dynamics of its sensors as well as a probabilistic description of the noise during measurements (Fang et al. 2008).

It has been said that in the history of statistical estimation theory, the Kalman filter is one of the most important discoveries (Mohinder and Andrews 2014). It has enabled mankind to do many things that could not have been done without it.

B. APPLICATIONS

The Kalman filter has been used in many different applications. One of the key application for the Kalman filter is the control of complex and dynamic systems such as the following (Mohinder and Andrews 2014).

- aircraft
- ships

- spacecraft
- manufacturing processes

In a dynamic system, one has to understand what the system is doing in order to control it. In a complex dynamic system, it is not always possible or even desirable to measure every variable that is to be controlled. Hence, in order to estimate the inadequate information from the noisy and indirect measurements, the Kalman filter can be used (Mohinder and Andrews 2014).

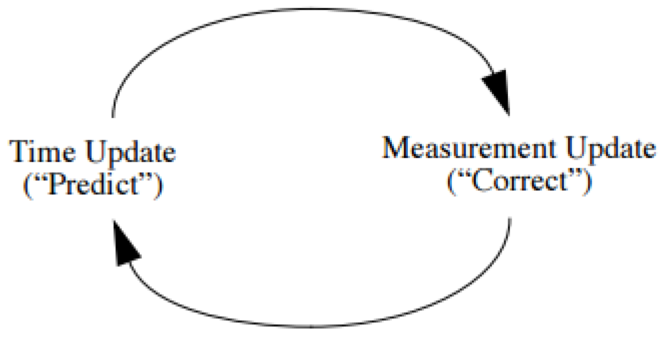
The Kalman filter has also been used in applications where people are trying to predict the future state of dynamic systems that are not within their control. Such dynamic systems could be commodity prices that are traded, celestial bodies' trajectories or the way the rivers flow in the case of a flood.

C. CONCEPTS

The Kalman filter is a state estimator that works by a combination of prediction and correction. To elaborate, the Kalman filter generates a conditional probability of the current state by first making predictions based on the dynamics of the system as well as the previous state. The prediction is later corrected using the measurements made by the system.

As depicted in Figure 3 the current state estimate is projected ahead in time by the time update, and this projected estimate is then adjusted by an actual measurement that occurs at that time in the measurement update.

Figure 3. Discrete Kalman Filter Cycle



From Greg and Gary 2006

1. State Estimator

The Kalman filter's main purpose is to attempt to estimate the actual state of a system, for example, the position of a mobile robot. More precisely, it estimates the state and provides an approximation of how inaccurate the estimate of the state is from the actual state. The estimation of the state is difficult because the state may change over time and it can be subject to noise.

2. Conditional Probability

Conditional probability is the robot's internal knowledge about its own state. As the state cannot be measured directly, the Kalman filter estimates the conditional probability of being in a state x_k given all the available measurements $z_1,.....,z_k$ and controls $u_1,....,u_k$. The probability of being in state x_k given the measurements $z_1,.....,z_k$ is called the estimate. We denote the estimate over the state of a variable x_k by $est\left(x_k\right)$, as shown in Equation 1.

$$est\left(x_k\right) = p\left(x_k \middle| z_1,....,z_k,u_1,....,u_k\right) \tag{1}$$

The estimate can be divided into the prior estimate and the posterior estimate, where the posterior estimate is incorporated after the measurement z_k. The Kalman filter calculates a prior estimate before incorporating z_k, just after incorporating the control u_k. The prior estimate is denoted as shown in Equation 2.

$$\overline{est}\left(x_k\right) = p\left(x_k \left| z_1, \ldots., z_{k-1}, u_1, \ldots., u_k\right.\right) \tag{2}$$

The prior estimate is the conditional probability of being at state x_k given all the measurements z up to and including step $k-1$. The posterior estimate is the conditional probability of being at state x_k given all the measurements z up to and including step k. Hence, in order to calculate the estimate, there is a need to formulate the functions for the system model $p\left(x_k \left| x_{k-1}\right.\right)$ and the measurement model $p\left(z_k \left| x_k\right.\right)$.

3. Prediction-Correction

In this section, the prediction and correction of the state by the Kalman filter is discussed.

(1) Prediction

The Kalman filter calculates the estimate by first computing the prior estimate before calculating the posterior estimate. The calculation of the prior estimate $\overline{est}\left(x_k\right)$ can be considered the prediction of the state of the system after a time step. The prior estimate tries to estimate the most likely state of the system after one time step without looking at the latest measurement information. This is done using the model of the system $p\left(x_k \left| x_{k-1}\right.\right)$ and the posterior estimate of what the state was in the last time step, $est\left(x_{k-1}\right)$.

(2) Correction

Prediction of the system state is bound to have errors due to noise in the system. As such, the prediction of the state of the system is likely to be different from the actual state. Hence, the calculation of the posterior estimate, $est\left(x_k\right)$, can be treated as the correction to the state estimate that resulted from the prediction. After the Kalman filter calculates the prior estimate, the new measurement data provides an indirect and noisy information of the actual state of the system. The new measurement can then be used to correct the predicted state. This is done by using the model of the measurement $p\left(z_k \middle| x_k\right)$. The model of the measurement describes how likely given a state x_k that the measurement results in the values z_k. Given the measurement data and the measurement model, the Kalman filter corrects the prior estimate in the state of the system.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYSTEM ENGINEERING OF ROBOT LOCALIZATION

This chapter uses a system engineering approach to examine the robot localization problem. First the problem of robot localization is defined. Next the boundaries of robot localization is determined. The limitations and constraints imposed on it is identified. The stakeholders of robot localization is then analyzed. This allows the requirements to be determined. The scope of the thesis is then defined before the operational concept is discussed. Functional analysis is then done on robot localization. Lastly the alternatives are considered and compared.

A. PROBLEM DEFINITION OF ROBOT LOCALIZATION

Essentially, the problem of robot localization is answering the question “Where am I?” from the robot’s perspective.

What this means is that the robot has to determine its relative position in an environment. This usually means determining its *x* and *y* coordinates as well as heading in a global coordinate system.

Localization is an important problem to solve as it is one of the key components in a successful autonomous robot. Without the robot properly determining its localization relative to its environment, it would be nearly impossible to decide what to do next.

Localization may seem to ask a simple problem, but solving it is almost never easy. In particular with a robot, localization is highly dependent on the characteristics of the robot. Techniques that work well for the robot in a certain environment may fail in another environment.

1. Problem

The localization problem can be categorized into three subgroups, which are defined by the information that is available to the robot initially and at run-time. The three subgroups are the position tracking problem, global localization problem, and kidnapped robot problem (Zhang et al. 2009).

In position tracking, it is assumed that the initial pose of the robot is known. Determining the position of a robot with reference to a known map of the surroundings is then achieved by factoring in the robot's motion to the robot's current state (Thrun, Burgard, and Fox 2005). On the other hand, global localization is a problem in which a robot has to determine its position after being randomly placed in an environment (Thrun, Burgard, and Fox 2005). Hence, the robot does not have information on its initial pose. It can be seen that the global localization problem contains the position tracking problem and is therefore a more difficult problem to solve (Thrun, Burgard, and Fox 2005).

Finally, another form of the global localization problem is the kidnapped robot problem. This is an even more difficult problem to solve. In this case, the robot may be translocated into another random position without the robot knowing. Under such a circumstance, the robot thinks that it knows where it is currently located, but the reality is that it does not. This makes it more difficult to solve than the global localization problem. While the robot might not be kidnapped in a real operation, its localization algorithm might fail. The ability to recover from failure can be measured by testing the localization algorithm on the kidnapped robot problem.

2. Uncertainty

As discussed in the previous section, robot localization can fail. A robot's localization failure can be attributed to a key element of robotics: uncertainty. Uncertainty arises from five possible factors (Thrun, Burgard, and Fox 2005), which are the dynamic environment, robot's sensors, robot's actuators, models of the surroundings, and algorithm computation.

The environment can create uncertainty as the physical world can be highly unpredictable, especially in a place where it is not purposefully planned out. While some environments can be structured in a way to reduce uncertainty, environments like homes and roads are highly dynamic.

The uncertainty from the robot's sensors is due to limitations in the sensor's perception. Firstly, the performance of sensors is bound by physical laws. For example, the range and resolution of the sensors are subjected to these laws, and sensors like cameras cannot see through walls. Secondly, sensors are also subjected to noise, which perturbs the measurements in unpredictable ways.

A robot's actuation produces uncertainty due to the use of motors, which are unpredictable to a certain extent. The lack of predictability can be contributed to wear and tear and control noise. This noise is even more significant in a low-cost robot whose motors are less precise.

A robot generate models to make sense of the surroundings. However, these models are an abstraction of the real world and are therefore inaccurate. This is because they are only a partial model of the underlying physical process of the robot and the environment.

A robot by its nature is a real-time system. This limits the number of computations that can be carried out per time unit. Hence, many algorithms for various functions are an approximation in order to achieve a timely response, which is a tradeoff for accuracy.

B. BOUNDARIES

In order to help scope the problem studied in this thesis, the boundaries of the problem have to be examined. There are many considerations when performing robot localization in an indoor and cluttered environment. It is only through studying the boundaries of such an environment that engineers can understand the complexity of the task.

There are, in essence, three basic classifications of boundaries. They are physical, functional, and behavioral boundaries. These boundaries lead to limitations and constraints, as well as boundary conditions of a system. Identifying and grouping the boundaries can help engineers better understand the problem and add more dimension to the perspective.

1. Physical

The physical boundary is determined by the limits of matter of one object. In robot localization, the physical boundaries can include parts of the robot or objects in its environment. Examples of physical boundaries can be seen in Table 1.

Table 1. Physical Boundaries

Physical boundary	Description
Fixed obstacles in an environment	Fixed obstacles prevent movement to certain areas. They can also block sensors' line of sight. These can be walls or objects like tables or boulders in an operational environment.
Dynamic obstacles in an environment	Dynamic debris are similar to fixed obstacles; however, their position or state can change during the course of the robot's operation. These can be objects that are moved around or doors that can be opened or closed.
Computers used in a robot	The computation demands of the robot determine the size and weight of the computer the robot uses. This can impact the power requirements, which, in turn, affect battery size and weight.
Robot's actuator	The actuator is the mechanism that affects the robot's movement. This leads to how the robot moves and the errors that the robot can make in movements.
Ground the robot traverses	The type of ground the robot is traversing determines what kind of mobility methods are used. This leads to types of errors made in movement.
Robots operating in the area	Other robots operating in the same area can behave like dynamic obstacles. They have the properties of an obstacle and are dynamic in that they can move around.
People operating in the area	People who are in the operating area can interfere with robot localization by acting like dynamic obstacles.
Buildings in an environment	Buildings are like fixed obstacles; however, the robot can operate inside or beside buildings. They have the properties of fixed obstacles. Buildings can impact things like the GPS signal, rendering it ineffective. Wireless communications can be negatively impacted by buildings by causing destructive interference.

2. Functional

The functional boundary is determined by two objects and their interfaces. A functional boundary is formed at the interface of objects. Through examining the functional boundary, the interactions between objects can be observed, as seen in Table 2.

Table 2. Functional Boundaries

Functional boundary	Description
Mobility of a robot	The robot can move using wheels, tracks, legs, or flight. This results in different types of interactions with the environment and, hence, different kinds of errors. If it is a ground robot, the traction with the ground is a factor for consideration.
Robot sensing its surrounding	The robot employs sensors to measure obstacles in its surroundings. The types of surfaces in the surrounding can cause different results. The robot's ability to sense its surroundings is necessary for it to successfully navigate its environment.
Computation of algorithms	The computation of algorithms to perform certain tasks is a boundary in the sense that there are limits to the computational power the robot can carry due to the fact that higher computation power means bigger, heavier processors and larger power consumption. This defines what kind of computational task the robot can take on.
Mapping of the environment	The function of mapping the environment is dependent on the task requirements of the robot. However, if mapping is necessary for the robot to perform its mission, then it sets certain conditions on the sensors. For example, the resolution of the sensor becomes critical to whether the data collected can be transformed into a map.
Use of a coordinate system	The robot's frame of reference is important when there is communication between multiple robots. The coordinate system has to allow the robot to communicate pose information with a common frame of reference in order for the robot to uniquely orient and locate the pose in discussion.
Modelling the physical world	Whenever the robot senses its surroundings using sensors, it has to model these surroundings based on the numerical data derived from the sensors. How accurately and precisely the robot can model the surroundings from the data affects the tasks the robot can do. For example, for the robot to simply avoid collision with a wall, it may not need a high fidelity model of the surroundings, but if it needs to plan a route through an area with debris scattered around, a higher fidelity model may be required.

3. Behavioral

Behavioral boundaries exist due to the existence and interaction between the physical and functional boundaries. Behavioral boundaries can be changed when the physical or functional boundaries are changed. These are outlined in Table 3.

Table 3. Behavioral Boundaries

Behavioral boundary	Description
Choice of robotics applications	Dependent on the accuracy of the robot localization, the choice of applications suitable for the robot can change. Robotics developers may only be restricted to certain application for a robot when the accuracy of robot localization is low.
Indecisive robot movement	If the robot localization data is inconsistent, the robot movement can appear to be indecisive. The robot's action is dependent on the data is has. The desired action of the robot can change rapidly if the data it obtains changes rapidly.
Missing or incorrect sensor data	If during the course of operation, the sensors are hit by some failure resulting in delayed data or incorrect data, the robot localizes itself using this data. The result is that the robot may think it knows where it is, but in fact, it is somewhere else. The robot acts on where it thinks it is at and, hence, behaves in a manner that is not expected.

C. LIMITATIONS AND CONSTRAINTS

This section discusses the limitations and constraints that are imposed on robot localization.

1. Limitations

Robot localization is plagued with several limitations that affect how successful a robot is in determining its location. Table 4 includes a discussion of the identified limitations.

Table 4. Limitations of Robot Localization

Limitations	Description
Uncertainty in environment	The environment is expected to be dynamic. Changes can occur to the environment after the robot has sensed and mapped the area. This dynamism of the environment poses a big challenge, as it burdens the sensor measurements with another inconsistency that has to be explained. For example, if the robot faces an open door that was previously modelled as closed, the robot has to decide if the environment has changed or if it is not where it is supposed to be. Frequently in robotics, the world is assumed to be unchanging, with the robot itself the only thing that varies over time (Thrun 2002).

Uncertainty in sensors	Sensors measure an approximation of what actually is. This approximation is bound to include errors. The range and resolution of the measure is also subject to the laws of physics. The data that the sensors provide are discrete numbers; hence, there will be truncation of the actual measurements that took place. The robot can only sense what it can sense. In a complex environment, there are frequently parts of the surroundings that are occluded from the robot's sensors, thereby reducing the information the robot has to work with.
Uncertainty in a robot	The actuators that move the robot are also unpredictable to a certain extent. This is due to control signal noise and wear and tear. For a low-cost robot, this can be expected to be even more prevalent. Therefore, it is not possible to track a robot's position by only keeping track of the movement commands that have been issued to it.
Uncertainty in models	Localization computation is based on certain models of the sensors and robot. The models are an abstraction of the real world, and it is only possible to partially model the underlying physics of the robot and its environment.
Uncertainty in computation	For a robot to be functional, it has to be a real-time system. This limits the amount of time the robot has to process all the information that it consolidates from its sensors. In order to limit the time of processing the information, it has to limit the amount of computation that can be carried out. As such, there is generally a sacrifice of accuracy in order to achieve a timely response.
Statistically dependent errors	If measurement errors are statistically independent, the effects of errors can be negated simply by taking a lot of measurements. However, in robot localization, the errors in estimating location are statistically dependent. This is due to the fact that errors that are made in the past compounds with the errors made now. Hence the errors accumulates over time.
High dimensionality of problem	In a two-dimensional representation of the environment, a robot requires thousands of numbers. When it comes to a three-dimensional representation of the environment, there are easily millions of numbers the robot has to track. Statistically speaking, every one of the numbers adds a dimension to the fundamental problem.

2. Constraints

In robot localization, due to the type of robotics application, there are design constraints on the robot hardware as well as the software. These constraints on the robot in turn impose constraints on the robot localization, be it algorithm complexity or supporting hardware like sensors. These are examined further in Table 5.

Table 5. Constraints of Robot Localization

Constraints	Description
Algorithm complexity	Robot localization can occur in many forms, some more computationally intensive than others. However, in many robotics applications, onboard processors are limited by space, weight, and power constraints; hence, localization algorithm complexity is a constraint in terms of computational load.
Lack of external supporting infrastructure	For a robot to be truly mobile and not be confined to a fixed site, the subsystem affecting the robot's ability to perform localization has to be onboard the robot. This imposes the constraint that the localization technique cannot use any external sensors or infrastructure.
Lack of GPS support	In many robotics applications, the area of operations is frequently indoors or under a canopy. As such, the signal from the GPS is unable to reach the robot. In cases where the area of operations is an urban environment, even if the robot is not indoors, the presence of many tall buildings will also deflect the GPS signal. When it comes to military applications, in wartime situations, the GPS signal can also be jammed. Hence, by placing the constraint of not using GPS in the robot localization, it creates a more robust system that is not constrained to outdoor operations.

D. STAKEHOLDER ANALYSIS

Stakeholders are any party that have a right, claim, or share in a system. The characteristics of the system affects the party's needs and expectations. In this section, we identify the stakeholders in robot localization and describe their needs.

1. Robot Developers

The first stakeholders in robot localization are robot developers. This group is responsible for developing a variety of robot types, including mobile factory robots, search and rescue robots, military urban robots, and military robotic pack mules.

a. Mobile Factory Robot

For robot developers developing a mobile robot for use inside a factory, the challenge to robot localization is that the robot is operating indoors. As such, the robot does not have access to a GPS signal that can help it determine its position. While it is possible to place markers around the factory that the robot can use to locate itself, there is a limit as to how pervasive the markers are allowed to be. There will likely be a need for a robot to track its own positions between markers. External sensing is an option to track robot positions. However, such a sensor infrastructure can be very expensive, and it may not be scalable.

In a factory setting, it can be expected that there will be multiple robots working within line of sight of one another. This provides a setting where robot collaboration is possible. Even if the robots' exact position with respect to the factory is not known, knowing their position with respect to other robots can still be very useful. This allows the robots to prevent collisions with one another. It also allows for the robots to move in formation.

For the robot to be truly free to roam within the required area of operations, some form of robot localization technique where external sensors are absent has to be developed.

b. Search and Rescue Robot

For developers of a search and rescue robot, one of the environments that has to be considered is an urban indoor environment. In the case of natural disasters or war, the likely places that the robot is expected to be deployed are highly populated urban environments. In such an event, buildings may have collapsed or been damaged, and there will be a need to search for survivors in such areas. It would be highly risky to send a human being into such an environment; hence, deploying a robot would be desirable.

Again, once the robot navigate into the building, it will lose access to GPS signals. The robot would have to rely on itself to perform localization. It would be a highly dynamic environment with no prior maps in existence. It is not possible to have any existing sensing infrastructure to help the robot localize. In order for the robot to operate, a robot localization technique would have to be developed.

In a search an rescue environment, there are many tight spaces and the robot face the likelihood of being damaged or destroyed; therefore, it is desirable to deploy many cheap and small robots instead of a single large, expensive robot. In order for the multiple cheap robots to perform complex tasks, collaboration between the robots is necessary and desirable.

Similar to the factory robot, even if the search and rescue robot are unable to figure out its exact pose with respect to the surrounding environment, knowing its relative position with reference to other robots is useful. This allows the robots to effectively spread out, covering more areas as well as preventing collisions from taking place.

c. Military Urban Robot

In urban military operations, a robot can be very useful. The urban environment is an important operating arena for the military, yet it can be very dangerous for soldiers. In urban combat, a robot can be deployed to perform surveillance inside buildings where it would be dangerous for a soldier to enter. In such a scenario, the robot is entering an unknown operating area, and there is no access to a GPS signal.

Yet, in order to ensure that the robot's coverage of the operating area is sufficient, there is a need to estimate the position of the robot so it is not congested with other robots in any particular spot. In addition, when there is a threat detected, there is a need to identify the last known position in order for the operator to react to the situation.

d. Military Robotic Pack Mule

In military operations, a robotic pack mule can be very useful in helping soldiers carry more equipment without weighing down the soldiers. The robotic pack mule is designed to operate in different types of terrain and environments. One of the environments can be under a forest canopy where there is no line of sight to GPS signals. Yet for the robotic pack mules to move in formation, there is a need for the robots to estimate their position with respect to other robotic pack mules in the vicinity.

2. Navy Fleet

In a Navy fleet, there is access to GPS in peacetime. However, during war, it can be expected that the fleet would have to operate under a GPS-denied zone. The ships may still be equipped with highly sensitive gyroscopes to try to track the ship's position. However, without a fixed marker, any small errors in the gyroscope would add to prior errors, and eventually the errors would be too big to be ignored.

Navy fleets usually operate with multiple ships. In a GPS-denied environment, it would be beneficial in terms of both navigation and weapons targeting if the ships were able to utilize one another's data to further reduce errors in localization.

E. REQUIREMENTS ANALYSIS

From the various stakeholders and their needs that were defined in the previous section, it can be seen that while the types of operations and the goals are different, there are similarities in the limitations and conditions. By studying their needs using a robot localization technique, a set of requirements can be formulated. In addition, it is clear that robotics is one of the key areas that can benefit extensively from localization.

For an individual robot, localization can be achieved by many means. Without external help with localization, the robot can attempt to keep track of its pose by measuring wheel rotation, using inertial navigation systems (INS), measuring the optical flow of the ground, etc. However, in all these methods, there is no means of zeroing out any drift or errors compounded over time, no matter how small the errors. However, in the case of multiple robots collaborating on a task, knowing their position in reference to other robots is still useful.

By not having external infrastructure to help with the task of localization, each robot has to take up the computation load. The processing power of a mobile robot is always a limitation due to weight and power constraints. As such, localization techniques that do not require heavy computation is desirable.

From the perspective of the various stakeholders discussed previously, the requirements for the robot localization technique include:

- Provide robot localization data with fewer errors than individual robot odometry would produce.
- Provide relative position with reference to other robots.
- Provide timely localization data.
- Minimize computation requirements for generating results.

F. SCOPE

Scope defines the degree to which the project's goal or purpose covers the boundary. It is measured by tasks that will satisfy the stakeholders of the project. The scope of the outcome of the project is defined by its physical, functional, and behavioral boundaries.

This effort provides focus and ensures that any solutions developed will enhance the capabilities of the system.

(1) Within Scope

The scope of this thesis is to identify how, in the case of multiple robots, localization can be improved. Key parameters are to be identified that affect the performance of the localization. In this case, the sensors that facilitate localization are expected to provide imperfect measurements; hence, the sensors' performance is factored in the analysis.

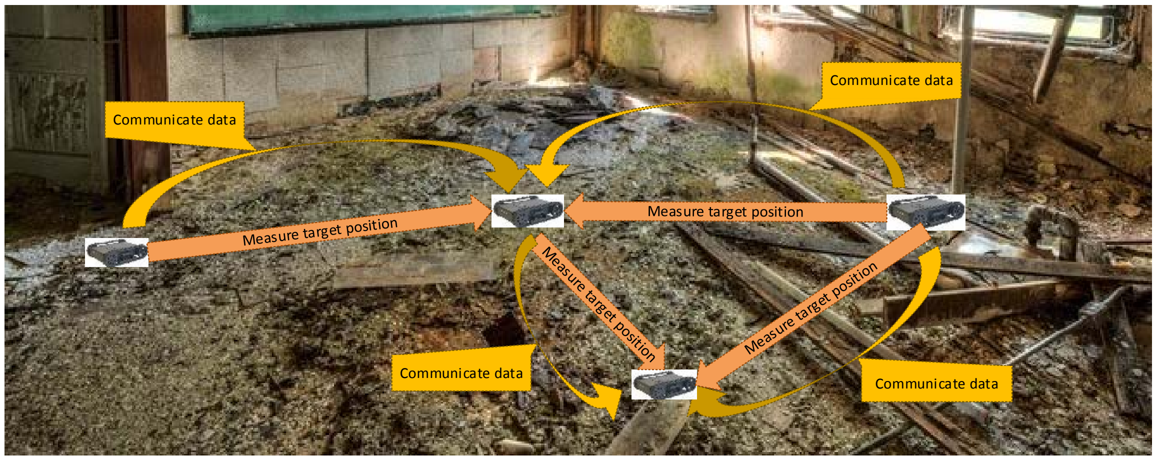
(2) Outside Scope

This thesis does not include the development of the robot and implementation of the sensors that facilitate the localization. It is assumed that the robot is able to provide the input parameters to facilitate the localization algorithm. This thesis also does not include any mission that the robot might undertake. That is, there will not be any development of algorithms specific to the accomplishment of any particular task other than localization. As such, the robot is not expected to have any navigational algorithm. This means that the robot do not do course correction on its path.

G. OPERATIONAL CONCEPT

Ultimately, the robot localization technique developed would have to operate under the assumption that there are no external sensors, be it that the robot is operating indoors or in a GPS-denied area. The concept of operations for performing robot localization in a multiple robot collaborative manner is illustrated in Figure 4, each robot has an erroneous perceived pose of itself. Each robot makes a measurement of a relative position of all robots within the line of sight and then shares measurement data with them. Each robot, armed with measurement data of its position, then computes its most likely position using a Kalman filtering technique.

Figure 4. Concept of Operations



Note: Multiple robots collaborating to measure one another’s position

With this set-up, there is no central control with regard to robot localization. As the robot measure and communicate with all other robots within its line of sight, this system is scalable. As the number of robots increases, it can be expected that the number of measurements taking place will increase and the accuracy of the localization is likely to improve.

H. FUNCTIONAL ANALYSIS

A system’s functions can be partitioned to provide more detail by performing a functional analysis. By delineating the functions, they can then be mapped into objects that can be built and integrated into the system. The output is a functional architecture, which is a hierarchical model of the functions performed by the system or its components.

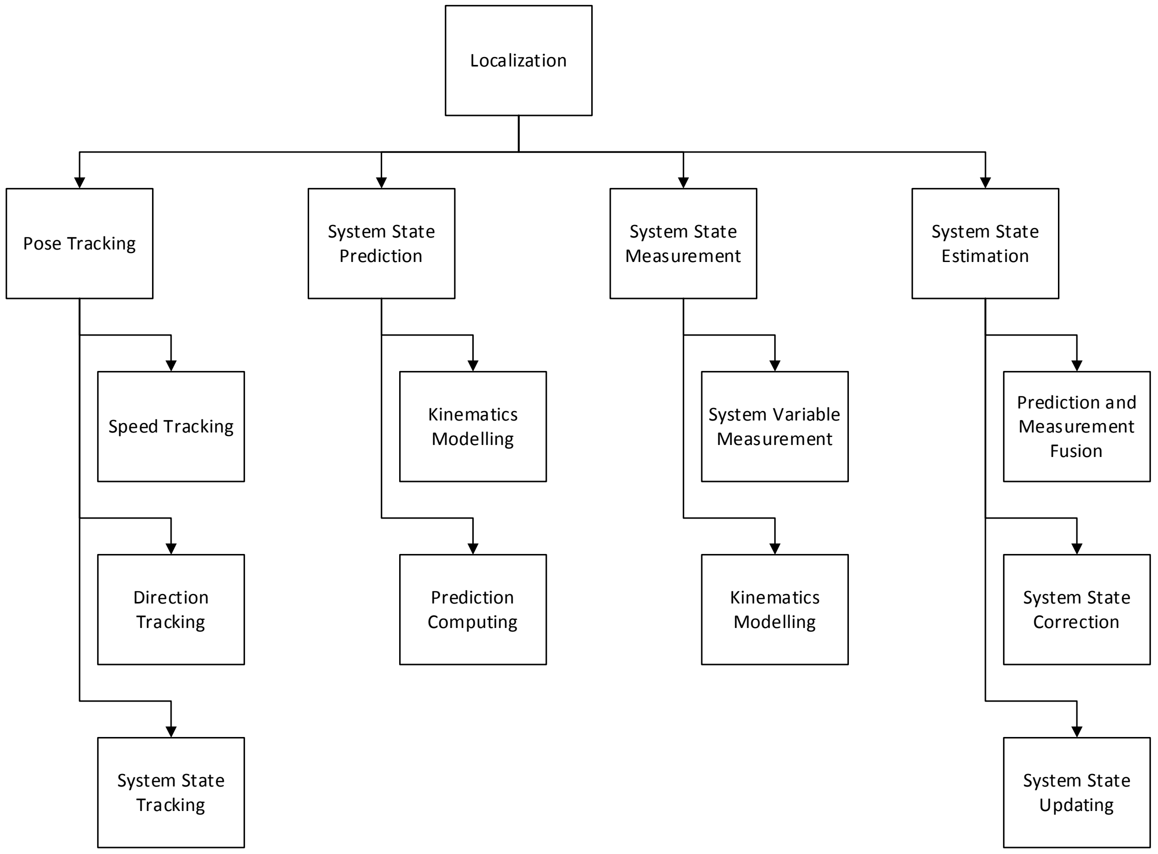
Here, the functional analysis is done to derive a functional decomposition of the robot operation. The top-level functions that constitute robot operations are defined along with the function of localization in order to give a view of where localization lies in a robot's operation. However, further functional decomposition is only done for localization, which is the focus of this thesis. The functional decomposition, and descriptions are listed in Table 6. The functional hierarchy of the function of localization is also depicted in Figure 5 to give a graphical view of the relationship between the functions.

Table 6. Robot Operations Functional Decomposition and the Corresponding Description (Focus is on Localization)

S/N	**Function**	**Description**
1.0	Robot operations	The top-level function of a robot system
1.1	Sensor measurement	To trigger sensors and take the readings
1.2	Goal tracking	To take stock of where the robot is with respect to the goal of the robot
1.3	Communication	To send and receive data between the robot and external entities
1.4	Actuation	To set in motion a part of the robot or the robot itself
1.5	Navigation	To get the robot from one point to another
1.5.1	Goal recognition	To identify where the robot should go to
1.5.2	Path planning	To find a way to get the robot to the desired point
1.5.3	Localization	To determine the most likely pose of self
1.5.3.1	Pose tracking	To take note of past pose and changes in pose
1.5.3.1.1	Speed tracking	To take note of past speed and changes in speed

1.5.3.1.2	Direction tracking	To take note of past direction and changes in direction
1.5.3.1.3	System state tracking	To identify system state from tracked parameters and take note of past state as well as changes in state
1.5.3.2	System state prediction	To compute the most likely state of the system based on the previous perceived state and the modelled changes of state
1.5.3.2.1	Kinematics modelling	To model the changes of the system state based on known information and a kinematics model
1.5.3.2.2	Prediction computing	To compute the predicted stated using the models generated
1.5.3.3	System state measurement	To compute the most likely state of the system based on measurements of system variables
1.5.3.3.1	System variable measurement	To determine a location based on measurements communicated to the robot by other observer robots.
1.5.3.3.2	Kinematics modeling	To model the system variables based on measurements and a kinematics model
1.5.3.4	System state estimation	To compute the most likely state of the system based on a combination of the predicted as well as measured system state
1.5.3.4.1	Prediction and measurement fusion	To merge the probability distribution function of the predicted and measured state to determine a state where the probability of the state is higher than either the predicted state or the measured state
1.5.3.4.2	System state correction	To correct the predicted current state to the state determined by the fusion
1.5.3.4.3	System state updating	To set the current state to the corrected system state

Figure 5. Functional Hierarchy for the Function Localization



I. ALTERNATIVES

There are many techniques to robot localization. Table 7 shows a few techniques that are used in many robotics applications. Each has its advantages and disadvantages.

Table 7. Types of Robot Localization

No.	Technique	Description	Advantages	Disadvantages
1.	Simultaneous Localization and Mapping (SLAM)	This is the process of computing the current position of an entity within a map that is being constructed and updated at the same time. There are many algorithms designed to perform this task.	The operation is not restricted to known areas, and it generates a map of the unknown surroundings.	It requires a great deal of computational power to both map and localize at the same time; hence, powerful processors and large amount of memory are needed, which leads to high power supply requirements.
2.	Global Positioning System (GPS)	This is a space-based navigation system. There are multiple GPS satellites in orbit and as long as 4 satellites are in the line of sight, the system is able to provide location information.	Location data do not drift as errors in measurements do not compound on past measurement errors.	This would not work indoors, under forest canopy, or in places with many high-rise buildings.
3.	Dead reckoning	This is the process of computing the current position by using a previously determined position while factoring in estimated movement over time.	It is not restricted to operation only in known areas.	Errors in measurement compound; hence, location data drift over time.

4.	Marker-based localization	This involves placement of beacons or markers in specific positions in the area of operation where the robot can identify and get a fix on its current position.	It does not require complex algorithms. Errors in localization do not compound and zero out whenever another marker provides new data.	It requires existing infrastructure; hence, the system cannot be operated in any unknown area.
5.	External camera motion capture	This involves an infrastructure of cameras placed around an operating area that can capture motion. A processor processes all of the captured motion and generates the location and trajectory data based on the entities in the area.	It provides very accurate position and even trajectory data.	It is expensive to set up and requires existing infrastructure; hence, the system cannot be operated in any unknown area.

IV. SIMULATION MODEL OF MULTI-ROBOT LOCALIZATION

This chapter discusses about the simulation model developed to examine the performance of data-sharing robot localization.

A. BACKGROUND

For a multiple robots collaborative localization technique, there are many connections and much cohesion and coupling between the robots. As the number of robots increases, the complexity of the system increases dramatically. As such, it becomes quite impossible for developers to analyze the performance of the localization technique.

However, in the development of collaborative robotics systems, there is a need to determine the required performance of individual robot in order for the collaborative robots team to produce the desired behaviors, in this case, the robot localization performance. For example, the choice of sensors affects the accuracy of the measurements, and the choice of mobility system affects the precision of the movement. The performance of these modules eventually affect how effective the collaborative robot localization is.

In the development of a multiple robots system, the cost and size of the robot are key factors to be considered in the effectiveness of the system. Hence, it is pertinent for developers to consider the tradeoff between the accuracy of data and the cost, weight, and power requirements. Using the best sensors can produce very accurate measurements that allow for accurate robot localization, but doing so would also make the robot expensive and physically large. It would render a multiple robots system designed in this manner unfeasible, especially when the anticipated operating environment is a tight urban space.

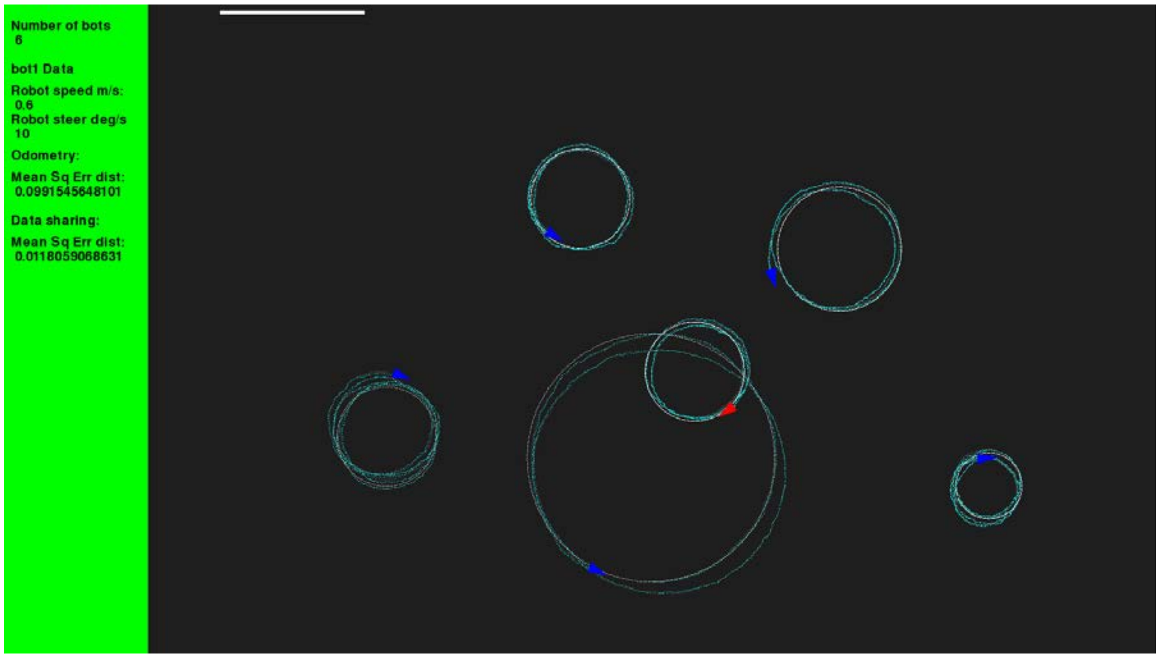
As such, there is a need for developers to analyze the performance of the robot localization by simulation before the development of the individual robot. Here a simulation is developed to model the behavior of the robot performing a collaborative robot localization technique. The simulation can take in different input parameters to represent different sensors or robots. The root-mean-square errors of the localization can be computed to determine the performance of the collaborative technique as compared to the performance when the robot individually handle its localization.

B. APPROACH

A simulation software was developed that models the movement of the robot by modelling the errors generated while moving. The individual robot odometry and data-sharing robot localization are simulated and the results of both can be compared. A Kalman filter model is developed here using kinematics equations and run in the simulation. The components of the simulation are discussed in detail in subsequent sections.

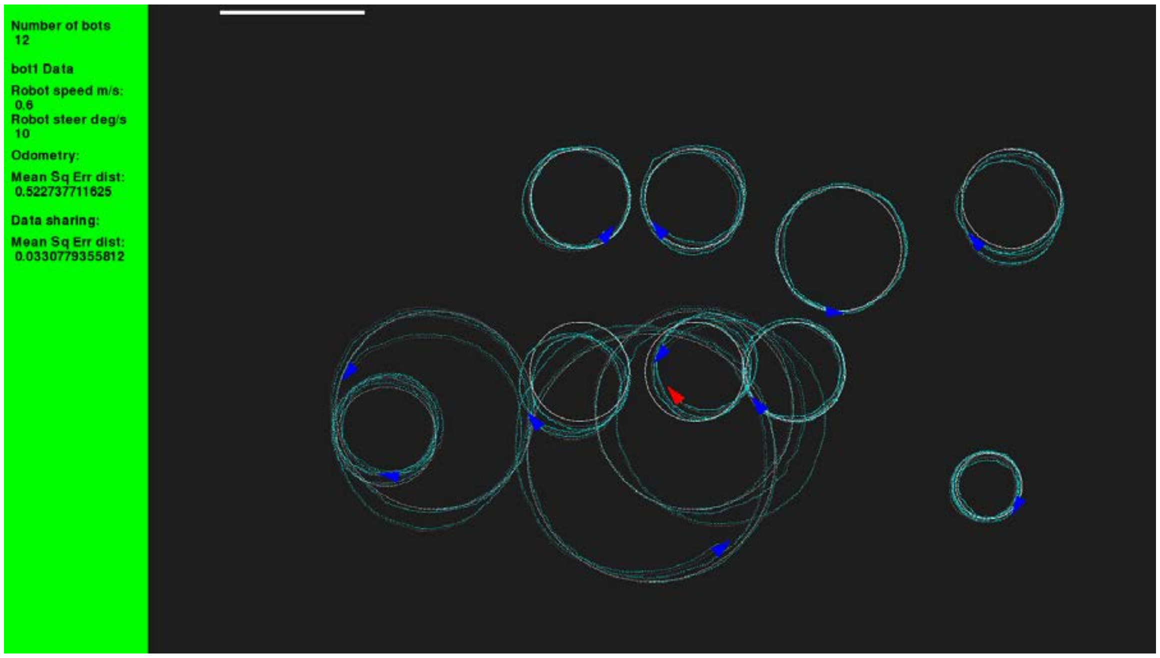
Figure 6 shows a snapshot of the actual computer graphics during the simulation with six robots. Figure 7 shows an example of simulation with twelve robots. In each simulation, the three key parameters are tracked. They are the actual position the robot has travelled, which is represented by the gray line, the perceived position from individual robot odometry, which is represented by the white line, and the perceived position from data-sharing robot localization, which is represented by the cyan line.

Figure 6. Simulation of Six Robots



```
Number of bots
 6

bot1 Data
Robot speed m/s:
 0.6
Robot steer deg/s
 10
Odometry:
Mean Sq Err dist:
 0.0991545648101

Data sharing:
Mean Sq Err dist:
 0.0118059068631
```

Figure 7. Simulation of Twelve Robots

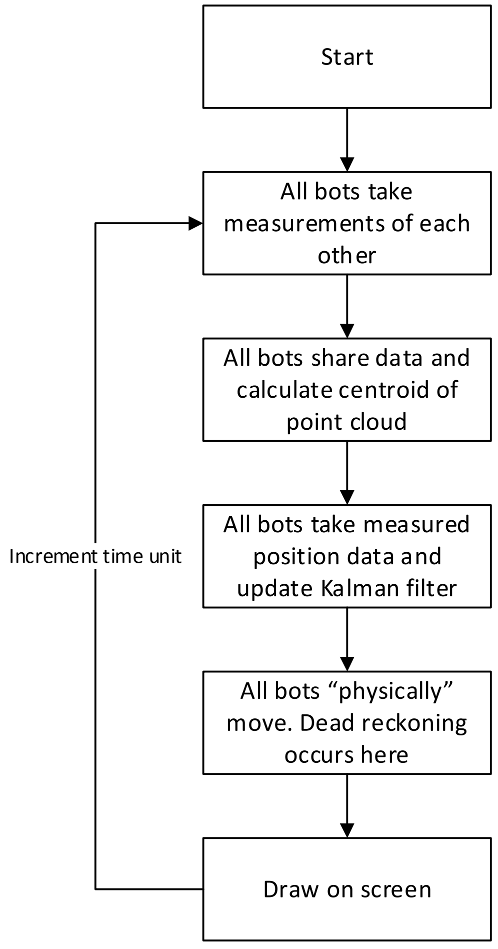


```
Number of bots
 12

bot1 Data
Robot speed m/s:
 0.6
Robot steer deg/s
 10
Odometry:
Mean Sq Err dist:
 0.522737711625

Data sharing:
Mean Sq Err dist:
 0.0330779355812
```

1. Simulation

For each time step, the simulation tracks and updates each simulated robot and generates each of the robot's localization technique's results. The two types of localization techniques are explored. The first technique is a simple individual robot odometry. The second technique involves each robot measuring one another's position and sharing these data. This allows generating a better estimate of robot's position. These techniques are outlined in Figure 8.

Figure 8. Flow Diagram of Simulation


Start
All bots take measurements of each other
All bots share data and calculate centroid of point cloud
All bots take measured position data and update Kalman filter
All bots "physically" move. Dead reckoning occurs here
Draw on screen
Increment time unit

During each simulation update, every robot updates its position based on its speed and direction.

The changes in the direction and position obey the Equations 3 through 7.

$$\phi_n = \phi_{n-1} + \frac{d\phi}{dt} \times T + \varepsilon_\phi \tag{3}$$

$$\theta_n = \theta_{n-1} + \frac{d\theta}{dt} \times T + \varepsilon_\theta \tag{4}$$

$$x_n = x_{n-1} + v \times \cos(\phi_n) \times \cos(\theta_n) \times T + \varepsilon_x \tag{5}$$

$$y_n = y_{n-1} + v \times \sin(\phi_n) \times \cos(\theta_n) \times T + \varepsilon_y \tag{6}$$

$$z_n = z_{n-1} + v \times \sin(\theta_n) \times T + \varepsilon_z \tag{7}$$

Where, ϕ is the yaw angle of the robot; θ is the pitch angle of the robot; x is the x coordinate of the robot; y is the y coordinate of the robot; z is the z coordinate of the robot; v is the speed of the robot; T the time interval between time steps; and ε is a Gaussian distributed noise.

It should be noted that Gaussian distributed errors are introduced to the movement of the robot as a way to represent slippage of the wheel with the ground, noise in the steer, skidding in the case of a ground robot, and wind in the case of an air robot.

The errors in movement are modelled as zero mean Gaussian distributed noise with a variance of $\sigma^2 = k^2 |D|$, where D is the distance travelled in meters and *k* is a constant representing the standard deviation of the error for every 1 meter travelled. A proper selection of *k* based on the specification of the robot is necessary for an accurate simulation.

As the error variance is factored by the distance travelled in the period of consideration, the speed of the robot as well as the time step of the simulation affect the amount of errors that are introduced to the movement. The faster the robot is moving, the greater the distance travelled in each time step. Similarly, the larger the time step between each calculation, the greater the distance travelled.

2. Individual Robot Odometry

Odometry is a form of navigational dead reckoning. It uses various kinds of sensors to estimate change in position over time. The current position can be calculated using a previously determined position and advancing it by the estimated position change. It is used by a wheeled or legged robot to estimate its relative position compared to the original location. Odometry is sensitive to errors due to it being subjected to cumulative errors.

Specific to this thesis, each robot tries to keep track of its pose individually. From Equations 8 through 12, it is noted that the robot perceive a perfect movement based on intended commands. However, the robot is unable to track the errors that occur in the actual movement.

The perceived pose is tracked by each robot using Equations 8 through 12

$$\phi'_n = \phi'_{n-1} + \frac{d\phi}{dt} \times T \tag{8}$$

$$\theta'_n = \theta'_{n-1} + \frac{d\theta}{dt} \times T \tag{9}$$

$$x'_n = x'_{n-1} + v \times \cos\left(\phi'_n\right) \times \cos\left(\theta'_n\right) \times T \tag{10}$$

$$y'_n = y'_{n-1} + v \times \sin\left(\phi'_n\right) \times \cos\left(\theta'_n\right) \times T \tag{11}$$

$$z'_n = z'_{n-1} + v \times \sin\left(\theta'_n\right) \times T \tag{12}$$

Where, ϕ' is the perceived yaw angle of the robot; θ' is the perceived pitch angle of the robot; x' is the perceived *x* coordinate of the robot; y' is the perceived *y* coordinate of the robot; z' is the perceived *z* coordinate of the robot; v is the speed of the robot; and T is the time interval between time steps.

As can be seen, the perceived pose tracking equation does not include the errors that occur in the movement. Hence, as the current perceived pose is built on the previous perceived pose, it can be expected that the unaccounted errors in movement would grow over time as the robot continues to move.

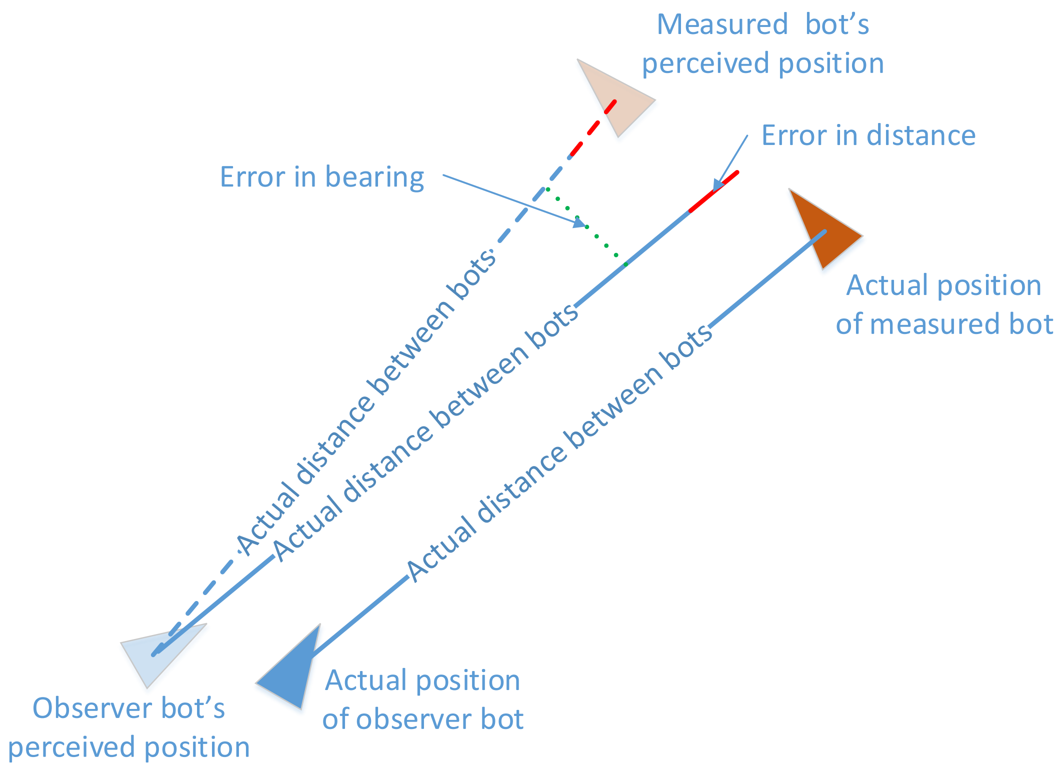
3. Data-Sharing Robot Localization

In the data sharing technique, each robot takes a measurement of all robots that are in its line of sight to estimate its position and share data. Each robot collects all the measurements of itself, which forms a point cloud.

Errors are introduced in the measurement of distance as well as bearing. This is to simulate errors in range measurement as well as the minimum angle resolution of the sensor. The errors increase as the distance between the robots increases, as the errors in bearing measurement resulting from the minimum angle resolution of the sensor result in a larger shift in perceived position. As can be seen in Figure 9 the small error in bearing measurement results in an error in the perceived position of the measured robot that grows as the distance between the two robots increases.

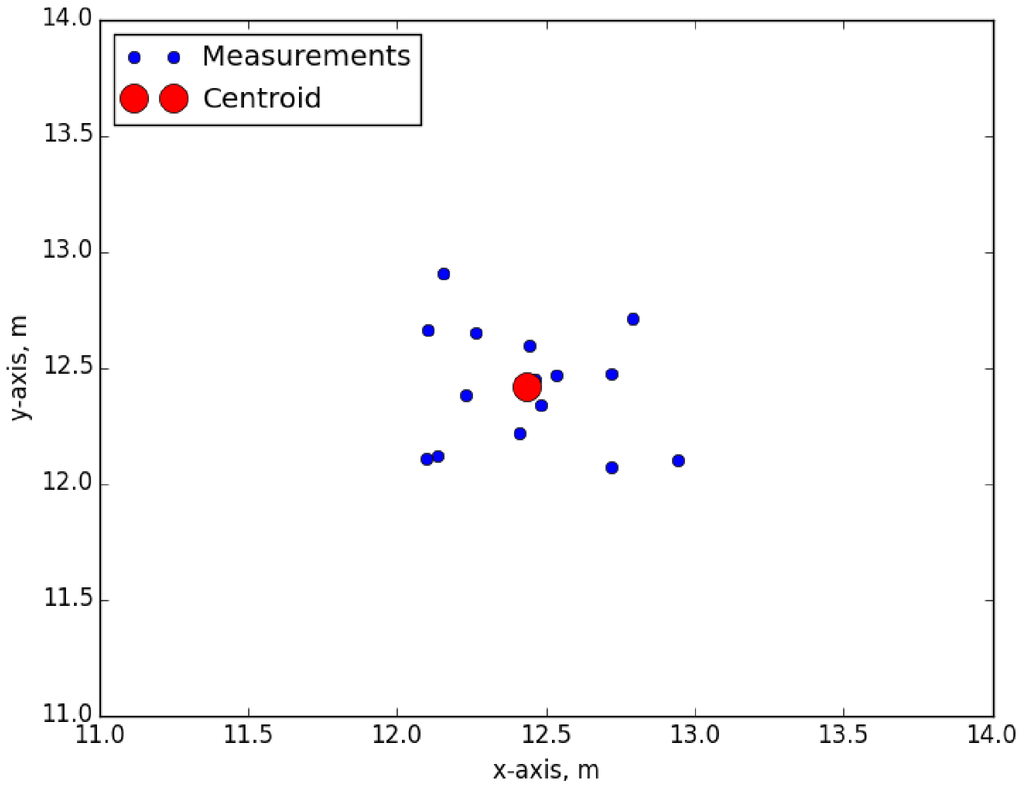
It must be noted that this error is compounded by the measurer's own erroneously perceived position.

Figure 9. Measurement Error Illustration


Measured bot's perceived position
Error in distance
Error in bearing
Actual distance between bots
Actual distance between bots
Actual distance between bots
Actual position of measured bot
Observer bot's perceived position
Actual position of observer bot

As mentioned, the measurements from multiple robots on one robot result in a point cloud where each robot gives a slightly different perceived position of the robot. From the point cloud, a centroid is computed to be used as the measurement. This can be seen in Figure 10.

Figure 10. Measurement Point Cloud Centroid Illustration



The estimated position from the data-sharing techniques are then fed as measurements for the Kalman filter model as described in the next section.

4. Kalman Filter Model

The Kalman filter model is based on the kinematic equations as shown in Equations 13 through 21.

$$x_k = x_{k-1} + \dot{x}_{k-1}T + \ddot{x}_k \frac{T^2}{2} \tag{13}$$

$$\dot{x}_k = \dot{x}_{k-1} + \ddot{x}_k T \tag{14}$$

$$\ddot{x}_k = \frac{\dot{x}_k - \dot{x}_{k-1}}{T} \tag{15}$$

Where T is the time delta between each time step; x_k is the *x* axis position at time step k (k = 1,2,3….); $\dot{x}_k$ is the velocity along the *x* axis direction; and $\ddot{x}_k$ is the acceleration along the *x* axis direction. Similarly,

$$y_k = y_{k-1} + \dot{y}_{k-1}T + \ddot{y}_k \frac{T^2}{2} \tag{16}$$

$$\dot{y}_k = \dot{y}_{k-1} + \ddot{y}_k T \tag{17}$$

$$\ddot{y}_k = \frac{\dot{y}_k - \dot{y}_{k-1}}{T} \tag{18}$$

Where y_k is the *y* axis position at time step k; $\dot{y}_k$ is the velocity along the *y* axis direction; and $\ddot{y}_k$ is the acceleration along the *y* axis direction.

For robots operating in three dimensions,

$$z_k = z_{k-1} + \dot{z}_{k-1}T + \ddot{z}_k \frac{T^2}{2} \tag{19}$$

$$\dot{z}_k = \dot{z}_{k-1} + \ddot{z}_k T \tag{20}$$

$$\ddot{z}_k = \frac{\dot{z}_k - \dot{z}_{k-1}}{T} \tag{21}$$

Where z_k is the *z* axis position at time step k; $\dot{z}_k$ is the velocity along the *z* axis direction; and $\ddot{z}_k$ is the acceleration along the *z* axis direction.

This model uses kinematic equations in all three axes in the three-dimensional world. This allows the simulation to include scenarios that models a flying robot or robots operating at different heights. If the scenario to be simulated is for ground robots all operating on the same plane, then the *z* axis can be ignored and set to 0.

With kinematic equations for all three axes, the state equation of the proposed model is shown in Equation 22.

$$\begin{bmatrix} x_k \\ \dot{x}_k \\ y_k \\ \dot{y}_k \\ z_k \\ \dot{z}_k \end{bmatrix} = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ \dot{x}_{k-1} \\ y_{k-1} \\ \dot{y}_{k-1} \\ z_{k-1} \\ \dot{z}_{k-1} \end{bmatrix} + \begin{bmatrix} \frac{T^2}{2} & 0 & 0 \\ T & 0 & 0 \\ 0 & \frac{T^2}{2} & 0 \\ 0 & T & 0 \\ 0 & 0 & \frac{T^2}{2} \\ 0 & 0 & T \end{bmatrix} \begin{bmatrix} \ddot{x}_k \\ \ddot{y}_k \\ \ddot{z}_k \end{bmatrix} \tag{22}$$

With the model defined in this state equation, the measurement is incorporated into the Kalman filter using Equation 23.

$$\begin{bmatrix} Z_{x_k} \\ Z_{\dot{x}_k} \\ Z_{y_k} \\ Z_{\dot{y}_k} \\ Z_{z_k} \\ Z_{\dot{z}_k} \end{bmatrix} = H \begin{bmatrix} m_{x_k} \\ m_{\dot{x}_k} \\ m_{y_k} \\ m_{\dot{y}_k} \\ m_{z_k} \\ m_{\dot{z}_k} \end{bmatrix} \tag{23}$$

Where $m_{x_k}, m_{y_k}, m_{z_k}$ are the observation data of the robot position; $m_{\dot{x}_k}, m_{\dot{y}_k}, m_{\dot{z}_k}$ are the observation data of the robot speed in the respective directions; $Z_{x_k}, Z_{y_k}, Z_{z_k}$ are the measurement data of the robot position; $Z_{\dot{x}_k}, Z_{\dot{y}_k}, Z_{\dot{z}_k}$ are the measurement data of the robot speed in the respective directions; and H is the observation matrix. The observation matrix H translates the observation vector into the measurement vector.

In the case of the simulation, as the observations of the robot's state are the actual measurement of the robot's state, there is no translation required; hence, H is just the identity matrix as shown in Equation 24.

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{24}$$

C. ASSUMPTIONS

Any simulation is an approximation of the actual physics of the real world due to the complexity. Hence, some assumptions have to be made to simplify the complexity. In addition, the focus of this thesis is on robot localization; therefore, the implementation of the supporting systems are not looked at in detail. This also leads to another set of assumptions. The following are the assumptions when performing the simulation:

- The robots have perfect communication with each other. There is no loss of or lag in communication.
- The robots all have synchronized time steps. This assumption can be mitigated in the real world by having the robots synchronize their clocks.
- There will be errors in individual robot odometry, which can be contributed by sources like errors in sensors, wheel deviation, and slippage. In the simulation, all errors are summed up and are assumed to be zero mean white noise.
- The algorithm and sensors are assumed to be developed because the focus of this thesis is on robot localization, and the development of the algorithm and sensors used by the robots for measuring each other's position is by no means an easy feat and could be an entire research project in itself.
- The line of sight of the sensors used for robot position measurement is assumed to be limited by range, but other robots will not obscure the line of sight.

Some of these assumptions are based on complexity that is difficult to address by computation, while others are due to time constraints of developing the simulation. As such, some of the assumptions can be addressed in future work in the simulation software.

D. SOFTWARE USED

The time-based simulation was created using the Python scripting language. It was used to model the robot and the coupling between the robots. Python was used because it is a general purpose high-level programming language, so it offers ease of development while allowing the programmer freedom to develop in any way.

Python's design philosophy emphasizes code readability as well as allowing the programmer to write fewer lines of code than other languages like C++. This helps future programmers who wish to further the development to quickly ease into the code.

Within the Python scripting language, the PyGame library is used to develop the graphical aspects of the simulation. The PyGame library is based on the Simple DirectMedia Layer development library.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESEARCH SCENARIO AND RESULTS

This chapter discusses about the performance metrics used to assess the data-sharing robot localization technique. The approach to the experiment is examined and a scenario is developed to test the technique. Finally, the results of the experiment are analyzed.

With the simulation software developed, this thesis proceeds to use the software as a tool to analyze various scenarios where multiple robots are deployed. The performance data of how well the robot track its position can be collected and compared. As the simulation software is developed to replicate the 3-dimensional space, it can be used simulate an air-based robot. By setting all metrics in the *z* direction to 0, the simulation software can also be used to simulate a ground-based robot. Here, the thesis conducted an experiment based on only ground-based robots.

A. PERFORMANCE METRICS

One of the key performance metrics is the root-mean-square errors of the perceived position as compared to the actual position. The metric can be used to compare the performance between a collaborative data-sharing robot localization with one where the robots all operate individually.

B. APPROACH AND ASSUMPTIONS

The simulation is based on a set of kinematic equations with noise introduced to the output at each time step. The input parameters and the amount of noise introduced are based on the scenarios considered. This is done by determining the specification of the robot and sensors in the scenarios and using the specifications and performance data as parameters in the simulation.

In each of these scenarios, multiple robots in collaboration are central to the robot localization. In each case, the robots are assumed to be mobile and able to communicate with each other. It is also assumed that the robots are able to take measurements of each other's pose. As previously mentioned in the scope of the thesis, the implementation of the robot is not considered in the thesis, and it is assumed that the robot can perform the tasks within the simulated specifications. For example, if the robot is using laser scanning rangefinders, it will be able to measure other robot's position with 1% error and up to a range of 30 meters.

Multiple Monte Carlo simulations are carried out for the scenarios to examine the performance based on different conditions. The root-mean-square error of the localization results against the actual position is tracked throughout the simulation. The errors in the positions at the end of each simulation are tracked. The root-mean-square errors are tabulated in Table 9 and Table 10. The following are some general parameters of the simulation:

Number of simulation runs = 1000

Time delta per time step in simulation = 0.1 seconds

Number of time steps per simulation run = 1000

C. AUTONOMOUS GROUND ROBOT SCENARIO

1. Scenario

In this scenario, several ground robots are placed in an indoor environment of 50 meters by 40 meters in size. The robots in this scenario are patrolling the area in a circular manner. Each attempts to track its own position. The robots simulated here are the Pioneer P3-DX, as seen in Figure 11. This model has a maximum speed of up to 1.2 meters per second. The Pioneer P3-DX is chosen for this scenario because it is a popular research mobile robot. As shown in Figure 12, the robots are retrofitted with the scanning laser rangefinder used to measure the relative positions of other robots.

Figure 11. The Pioneer P3-DX

Figure 12. Hokuyo UTM-30LX Scanning Laser Rangefinder

Two variations of the simulation scenarios involved different number of robots. Specifically, the two simulations involved six and twelve robots respectively. This was done to examine the effect of the number of collaborating robots on the accuracy of the robot localization.

2. Simulation Setting

The simulation allows for various input parameters to be tuned. In this experiment, 3 key parameters are looked at: the number of robots, the sensor performance, and the individual robot odometry performance. This is to identify where and how data sharing can help in robot localization.

The justification for the various parameter values used in the experiment configurations is discussed here. For the sensor, the scanning laser rangefinder, the measurement performance is as follows:

Range:	30 meters

Angular resolution:	0.25 degrees
Accuracy:	+/- 30 millimeters

For the individual robot odometry, the performance of the robot is explored with a variation of odometry error parameters. The errors are modelled as zero mean Gaussian distributed noise with variance modelled as $\sigma^2 = k^2 \left|D\right|$ where *D* is the distance travelled in the time step and *k* is a constant. The constant *k* represents the standard deviation of the error for every meter travelled. For this scenario, the following values of *k* where chosen.

k = 0.1, 0.01, and 0.5

The number of robots and the speed of each robot are also set as a parameter in the simulation. For this scenario, the effect of the number of robots is explored in the simulation. The following are the configurations:

Number of robots = 6 and 12

The simulation is conducted in various configurations. The configurations are summarized in Table 8.

Table 8. Summary of Configurations of Input Parameter to the Various Simulations Run

Input parameter configuration	**Sensor performance**	**Odometry performance, k**	**Number of robots**	**Robot speed**

1	+/- 0.03m acc
	30m range
		0.01	6	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s

2	+/- 0.03m acc
	30m range
		0.1	6	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s

3	+/- 0.03m acc
	30m range
		0.5	6	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s

4	+/- 0.03m acc
	30m range
		0.01	12	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s
				Bot7 = 0.6m/s
				Bot8 = 0.6m/s
				Bot9 = 0.6m/s
				Bot10 = 0.6m/s
				Bot11 = 1.2m/s
				Bot12 = 1.2m/s

5	+/- 0.03m acc
	30m range
		0.1	12	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s
				Bot7 = 0.6m/s
				Bot8 = 0.6m/s
				Bot9 = 0.6m/s
				Bot10 = 0.6m/s
				Bot11 = 1.2m/s
				Bot12 = 1.2m/s

6	+/- 0.03m acc 30m range	0.5	12	Bot1 = 0.6m/s
				Bot2 = 0.6m/s
				Bot3 = 0.6m/s
				Bot4 = 0.6m/s
				Bot5 = 1.2m/s
				Bot6 = 1.2m/s
				Bot7 = 0.6m/s
				Bot8 = 0.6m/s
				Bot9 = 0.6m/s
				Bot10 = 0.6m/s
				Bot11 = 1.2m/s
				Bot12 = 1.2m/s

These configurations are designed to explore the effects of the different performances of odometry and how data-sharing can improve robot localization. The configurations also explore how the number of robots can impact the performance of the data-sharing robot localization. Lastly, the thesis examines how data sharing can counteract the effect of erroneous robot localization from higher robot speed.

3. Results

The six different configurations described in the previous section are used to run the simulation, and the root-mean-square errors of the positions are collected for the data-sharing robot localization as well as the individual robot odometry.

Table 9 shows the results for running the simulation with six robots. Table 10 shows the results for running the simulation with twelve robots. This table only shows the data for the first six robots in comparison with the six robot simulation configuration.

Table 9. Root-Mean-Square Error of Data-Sharing Robot Localization and Individual Robot Odometry After 100 Seconds for a Six-Robot Simulation Averaged Among 1000 Simulation Runs

	k = 0.01		**k = 0.1**		**k = 0.5**	
	Data sharing	Odometry	Data sharing	Odometry	Data sharing	Odometry
Robot 1, v = 0.6m/s	0.245 m	0.0612 m	0.378 m	0.612 m	1.492 m	3.076 m
Robot 2, v = 0.6m/s	0.245 m	0.0607 m	0.379 m	0.618 m	1.493 m	3.087 m
Robot 3, v = 0.6m/s	0.245 m	0.0612 m	0.379 m	0.614 m	1.492 m	3.126 m
Robot 4, v = 0.6m/s	0.246 m	0.0608 m	0.379 m	0.610 m	1.494 m	3.070 m
Robot 5, v = 1.2m/s	0.246 m	0.0857 m	0.381 m	0.876 m	1.501 m	4.355 m
Robot 6, v = 1.2m/s	0.245 m	0.0898 m	0.379 m	0.864 m	1.498 m	4.291 m

Table 10. Root-Mean-Square Error of Data-Sharing Robot Localization and Individual Robot Odometry After 100 Seconds for a Twelve-Robot Simulation Averaged Among 1000 Simulation Runs

	k = 0.01		**k = 0.1**		**k = 0.5**	
	Data sharing	Odometry	Data sharing	Odometry	Data sharing	Odometry
Robot 1, v = 0.6m/s	0.105 m	0.0611 m	0.236 m	0.624 m	1.059 m	3.103 m
Robot 2, v = 0.6m/s	0.105 m	0.0623 m	0.236 m	0.624 m	1.059 m	3.128 m
Robot 3, v = 0.6m/s	0.105 m	0.0607 m	0.236 m	0.624 m	1.059 m	3.185 m
Robot 4, v = 0.6m/s	0.106 m	0.0609 m	0.237 m	0.624 m	1.059 m	3.068 m
Robot 5, v = 1.2m/s	0.106 m	0.0871 m	0.239 m	0.882 m	1.070 m	4.321 m
Robot 6, v = 1.2m/s	0.105 m	0.0879 m	0.238 m	0.869 m	1.069 m	4.422 m

As seen in Table 9 and Table 10, the improvement to robot localization by data sharing is not a constant across the different configurations of simulation. Table 11 shows the ratio of the data-sharing robot localization root-mean-square error against individual robot odometry root-mean-square error. This shows how data sharing affects the overall localization.

Table 11. Ratio of Root-Mean-Square Error of Data-Sharing Robot Localization Over Root-Mean-Square Error of Individual Robot Odometry

	Error reduction of data-sharing robot localization over individual robot odometry					
	k = 0.01		k = 0.1		k = 0.5	
	six robots simulation	twelve robots simulation	six robots simulation	twelve robots simulation	six robots simulation	twelve robots simulation
Robot 1, v = 0.6m/s	4.000	1.723	0.619	0.378	0.485	0.341
Robot 2, v = 0.6m/s	4.041	1.692	0.613	0.379	0.484	0.339
Robot 3, v = 0.6m/s	3.999	1.738	0.617	0.379	0.477	0.332
Robot 4, v = 0.6m/s	4.041	1.743	0.623	0.379	0.487	0.345
Robot 5, v = 1.2m/s	2.867	1.217	0.435	0.270	0.345	0.247
Robot 6, v = 1.2m/s	2.727	1.200	0.439	0.274	0.349	0.241

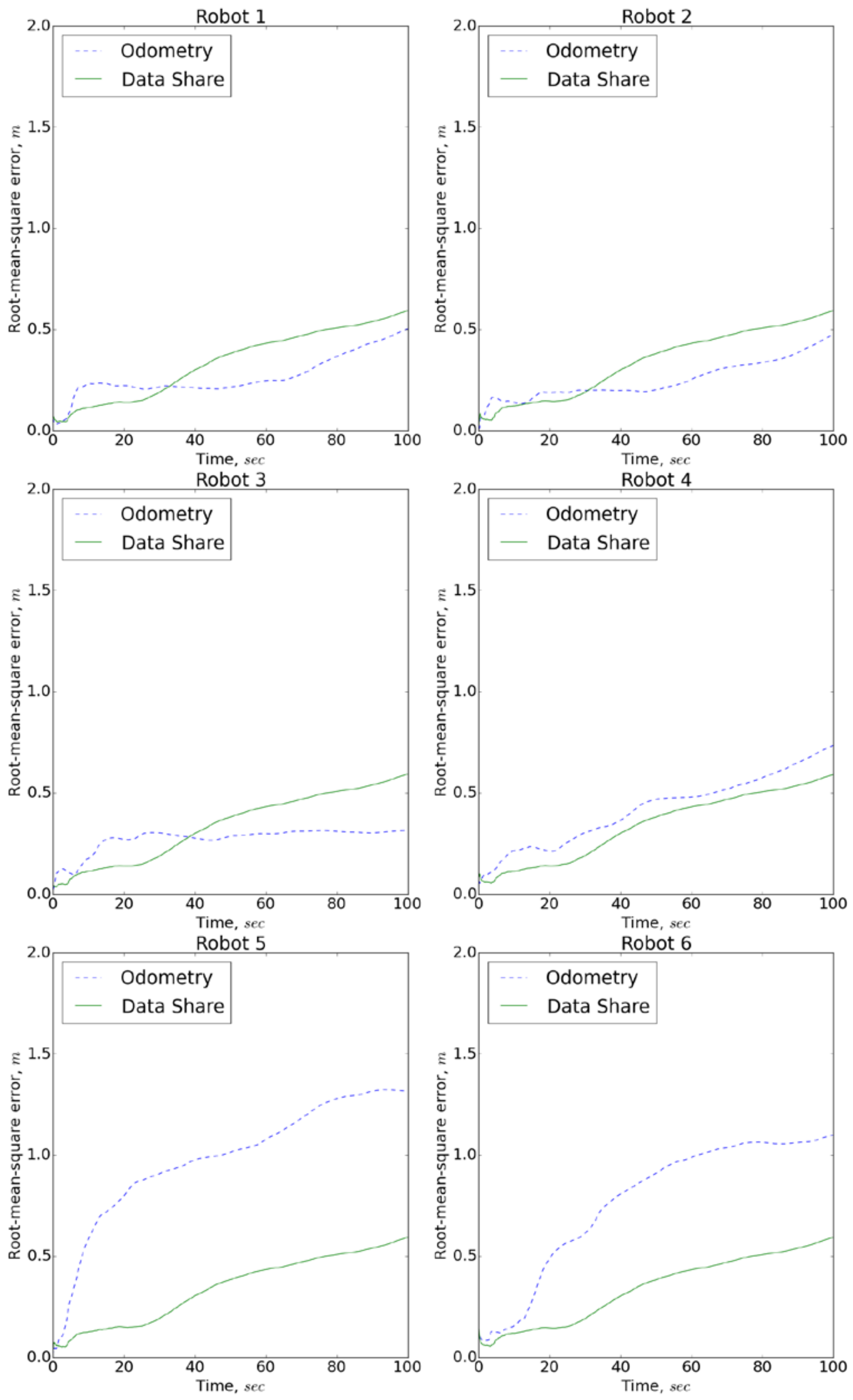
Note: The data of the above table is derived by dividing the root-mean-square error of the data-sharing robot localization by the root-mean-square error of the individual robot odometry.

Several observations can be made from the simulations conducted. They are as follows:

- Higher robot speed results in more errors in individual robot odometry due to the greater distance travelled in the time frame of the simulation.
- However, data sharing results in similar levels of error in robot localization regardless of the robot's speed.
- Data-sharing robot localization performance is worse than individual robot odometry when $k = 0.01$. Hence, if odometry errors are very small, measurement errors can cause more errors to localization via data sharing, and data-sharing robot localization should not be applied.
- However, when $k = 0.01$, the performance of data-sharing robot localization is better when there are twelve robots compared to when there are six. Therefore, having more robots sharing data leads to a decrease in measurement errors.
- Looking at robot 1, when $k = 0.1$, the ratio of errors is at 0.619. When $k = 0.5$, the ratio of errors is at 0.485. It can thus be observed that the benefits of data sharing become more significant as the odometry errors increase.

Previously, this thesis looked at the root-mean-square errors of the data-sharing robot localization at the end of the simulation. Next, this thesis looks at the performance of the data-sharing robot localization throughout the simulation. 0shows a single run of the simulation using input parameter configuration 2.

Figure 13. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 2 (Single Run Simulation)



As can be observed from 0due to the sharing of the position data among the robots, the data-sharing robot localization achieved similar root-mean-square error between all the robots, whereas the individual robot odometry shows different root-mean-square errors. While the root-mean-square errors fluctuate over the course of the simulation, it is noted that the errors generally grow over time for both data-sharing robot localization and individual robot odometry. However, data-sharing robot localization results in less errors overall.

Figures 14 through 19 show the root-mean-square errors of the simulation averaged over 1000 runs running the six different input parameter configurations. From the averaged data, it can be seen that all the errors are monotonically increasing. Even the data-sharing robot localization has an unbounded growth in root-mean-square errors. This is due to the lack of “ground truth,” which results in a drift of position data. Hence, over time, if the robot has no landmarks or any other means to zero out its errors during the operation, the errors in the localization data get too big to be ignored.

It is to be noted that aside from configurations 1 and 4 where the individual robot odometry errors are very small, the growth of error for data-sharing robot localization is significantly smaller than the individual robot odometry. The reduction in errors is more significant in the robots that are travelling at a higher speed. By travelling at a higher speed, the robots covered a greater distance within the time frame of the simulation, and therefore accumulate more errors.

Figure 14. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 1 (1000 Runs Simulation)

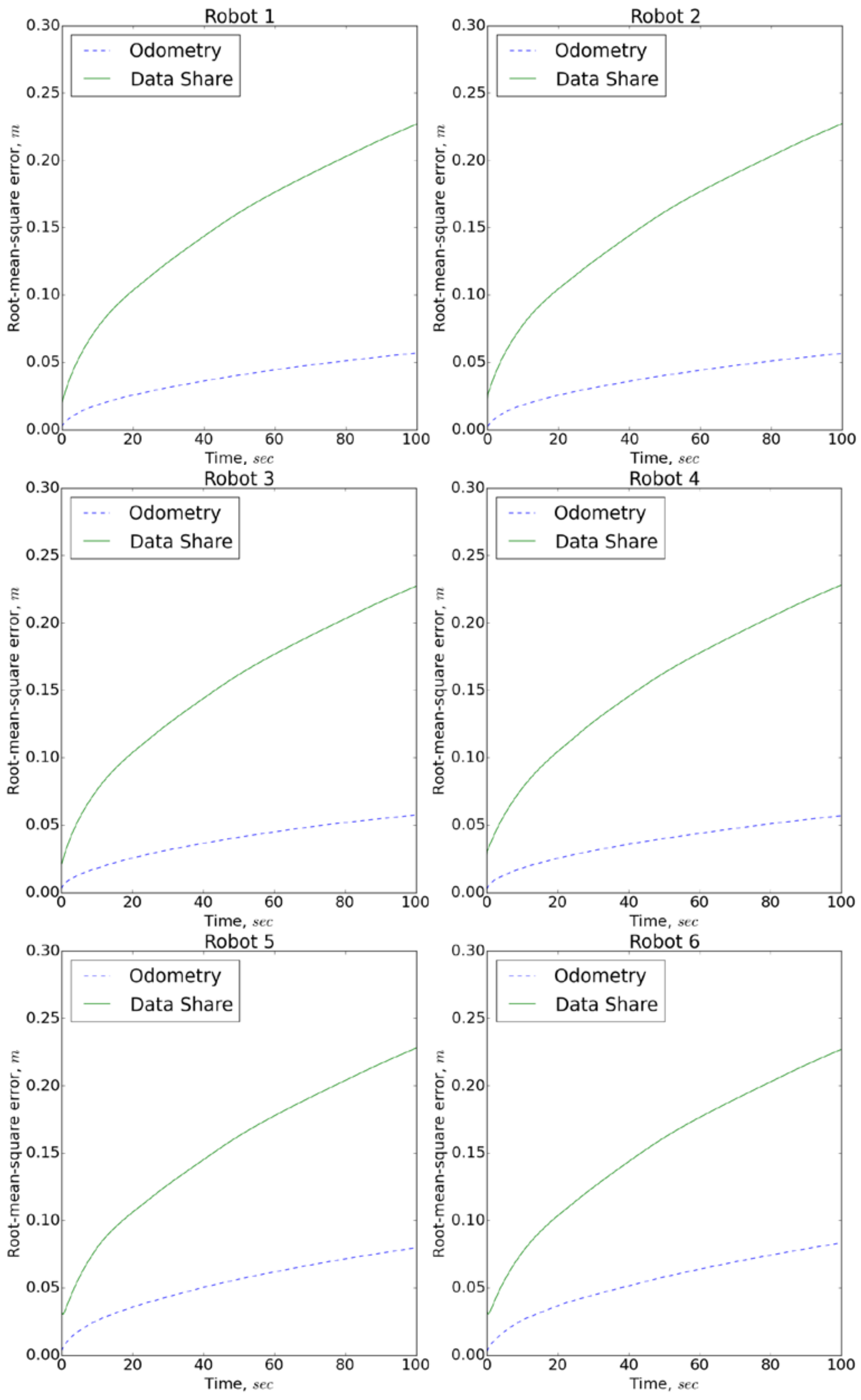


Figure 15. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 2 (1000 Runs Simulation)

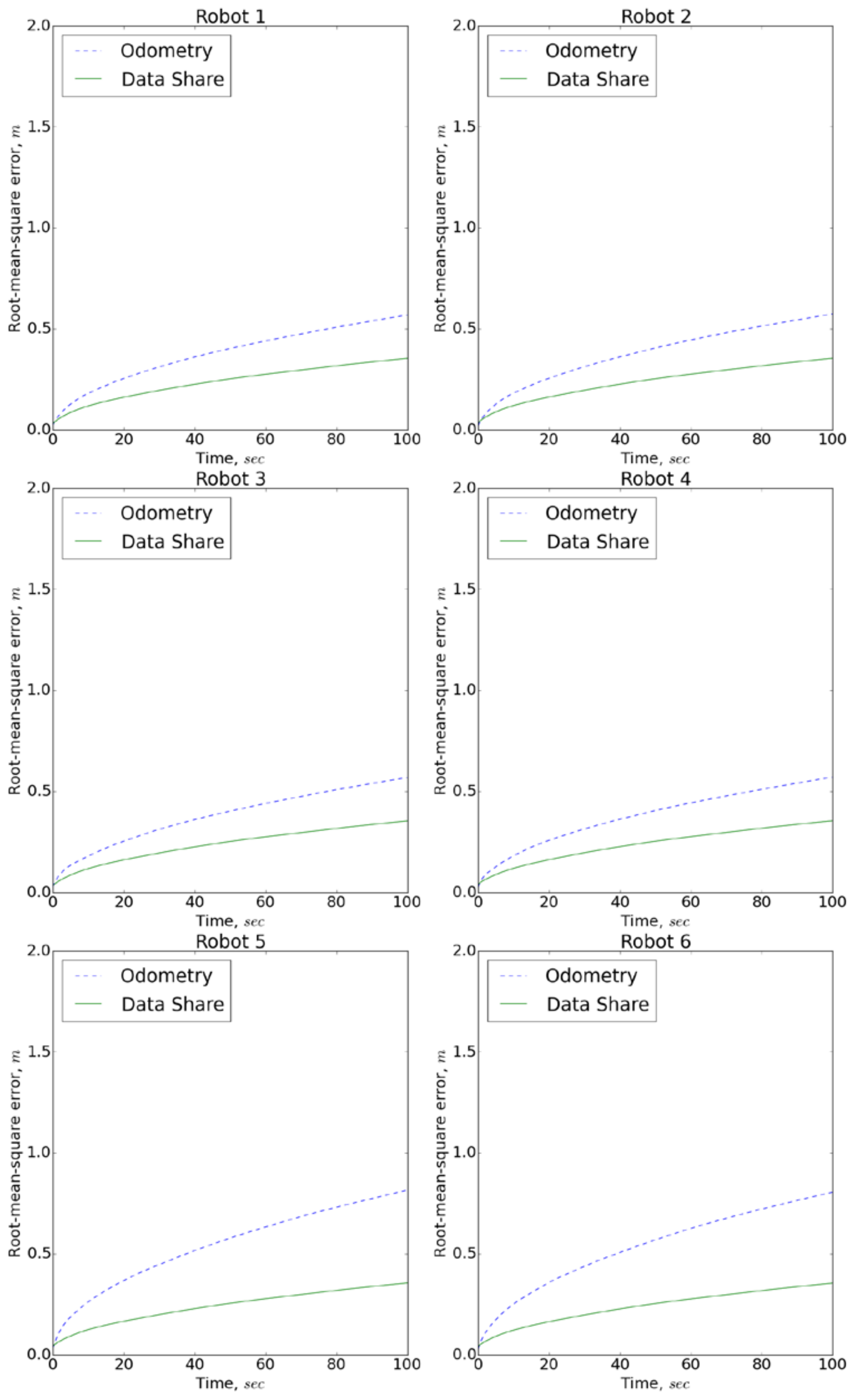


Figure 16. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 3 (1000 Runs Simulation)

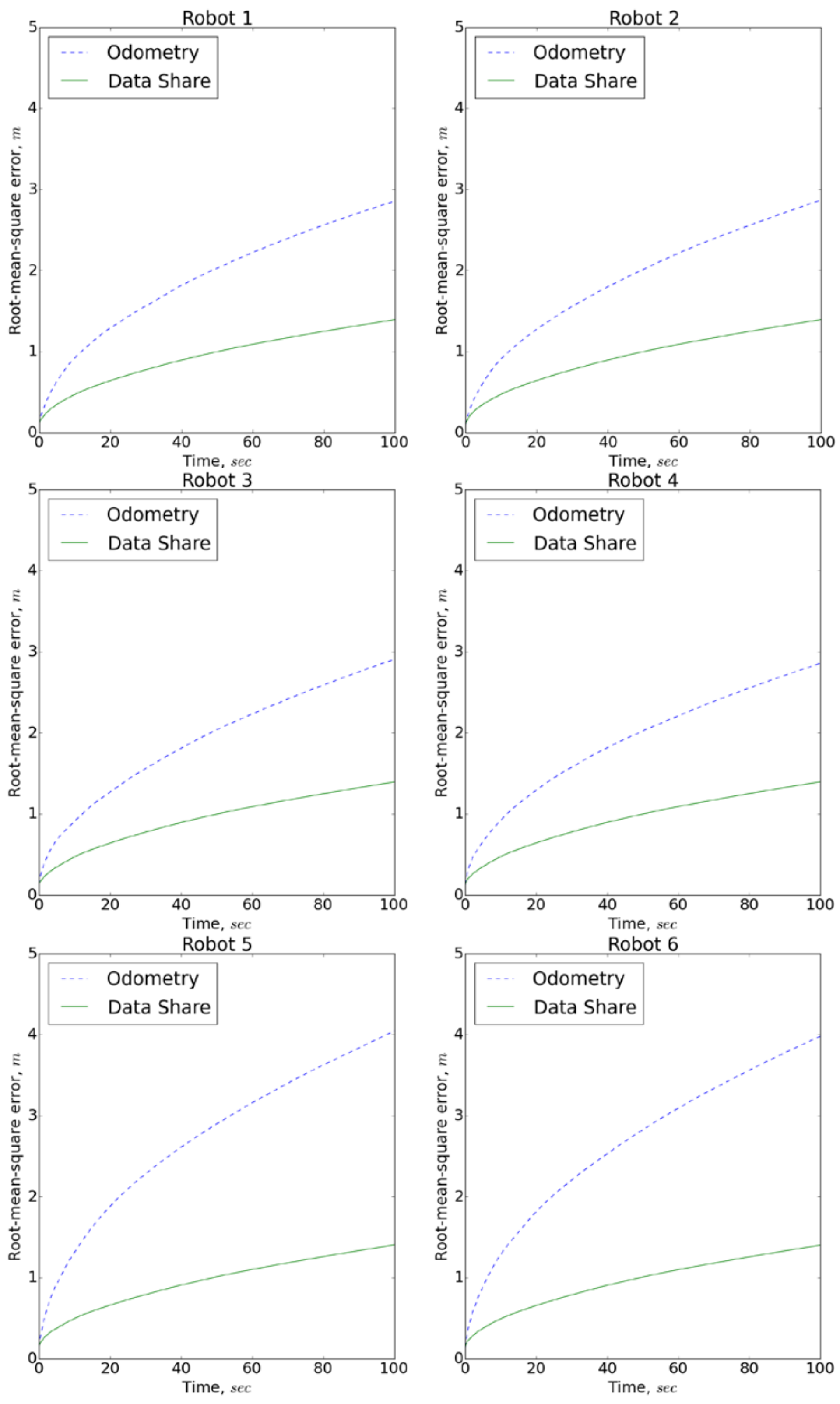


Figure 17. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 4 (1000 Runs Simulation)

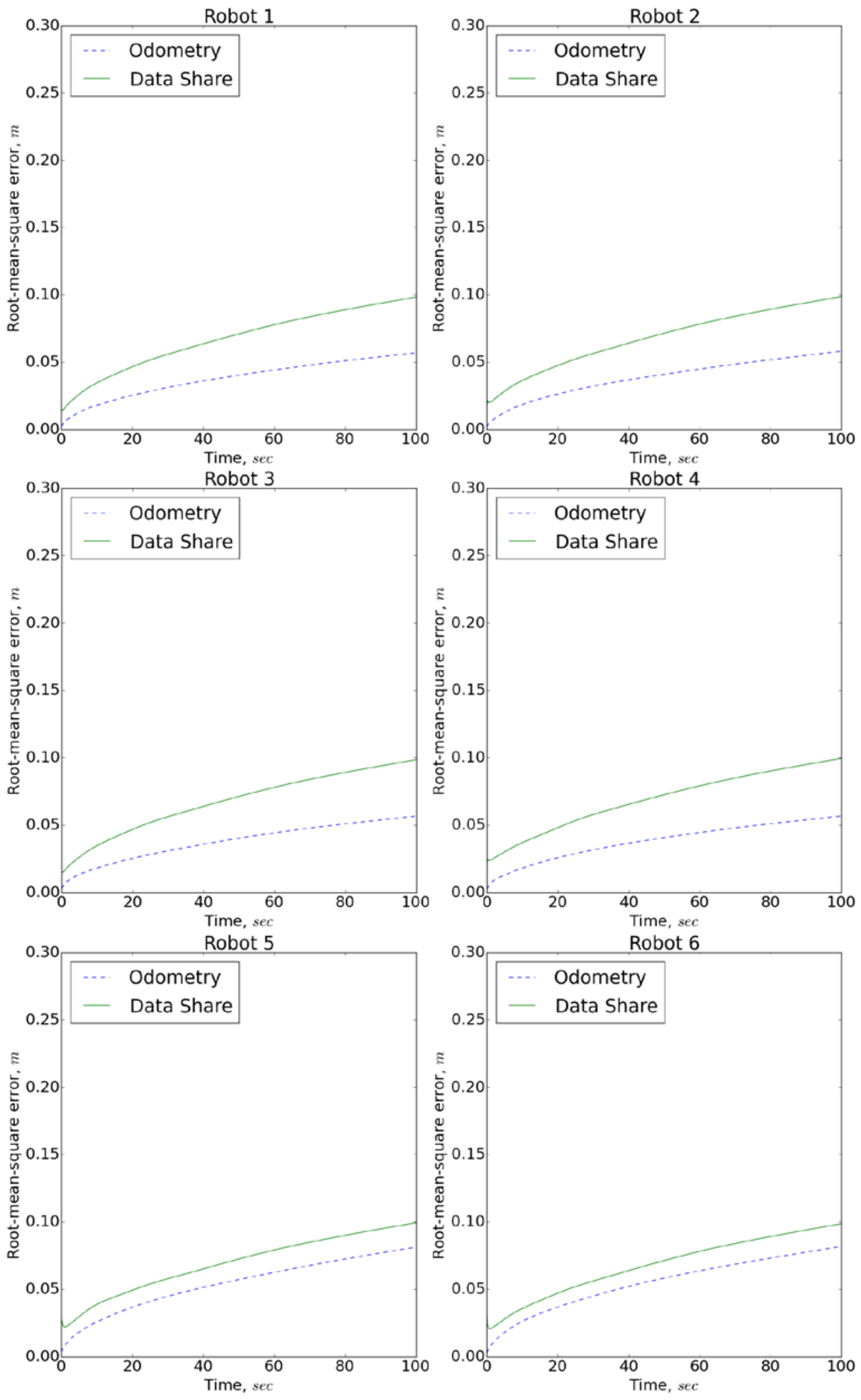


Figure 18. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 5 (1000 Runs Simulation)

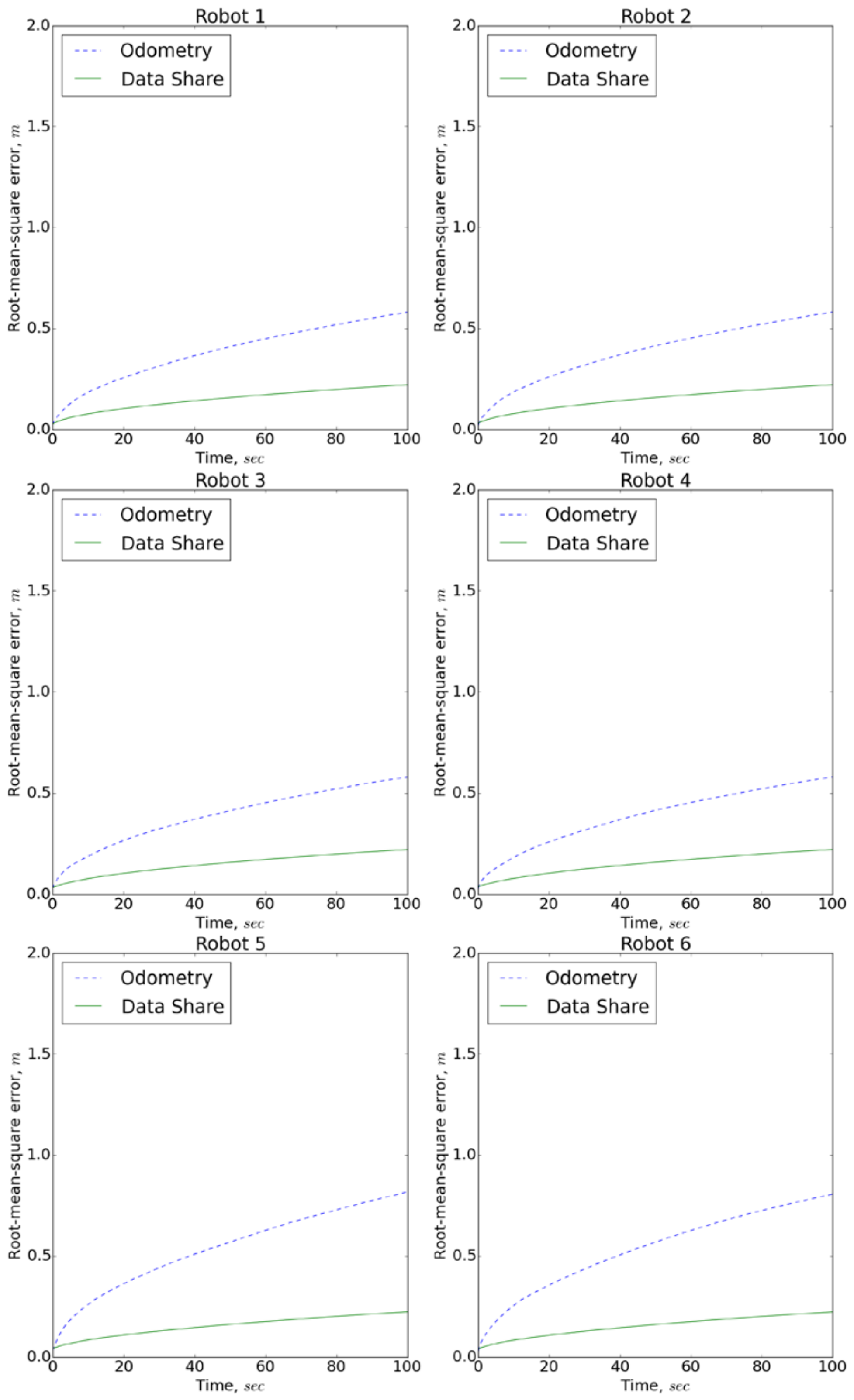
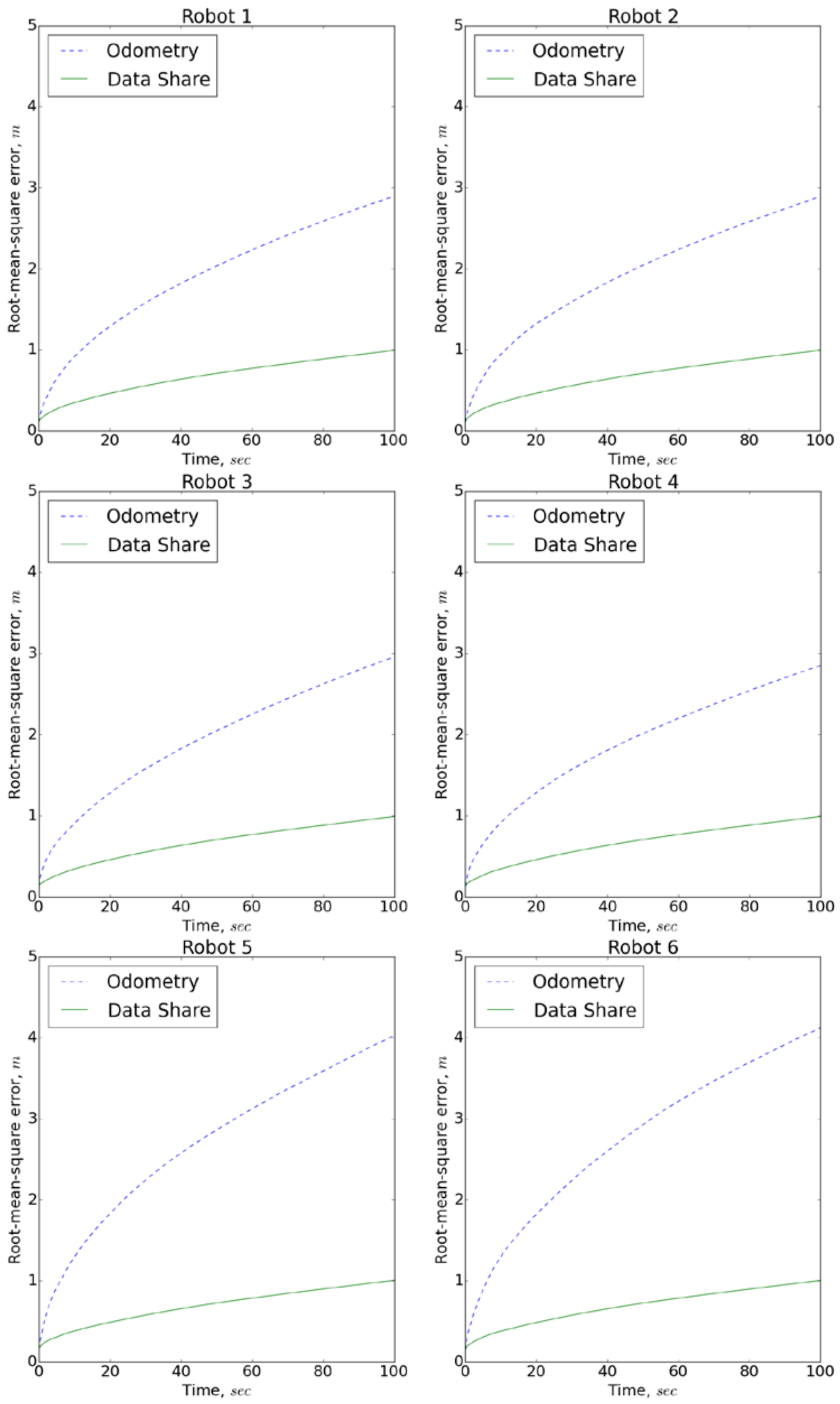


Figure 19. Root-Mean-Square Error for Simulation Running Input Parameter Configuration 6 (1000 Runs Simulation)



THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

In many cities, the danger of disaster to the population is real. Once disaster strikes, there are the issues of manpower constraint and hazardous areas of operation. Under such circumstances, the use of autonomous unmanned systems for search and rescue operations can free up valuable manpower and remove the need to send humans into dangerous areas. However, in order for autonomous unmanned systems to operate meaningfully, the systems have to be able to perform robot localization reliably. That is to say, the autonomous unmanned systems need to know their pose at all times and in a variety of environments they may encounter in a disaster area. This thesis introduces a data-sharing robot localization technique that significantly reduces errors in individual robot odometry when there is no external infrastructure to help provide "ground truth."

Before proposing a feasible robot localization technique, systems engineering techniques are used to analyze the problem statement as well as the stakeholders. This helps to further define the needs and requirements of robot localization. The boundaries of the problem are discussed, which helps to examine the limitations and constraints of robot localization. These discussions highlight the important considerations when designing a valid concept of operation. It is through the concept of operation that the data-sharing robot localization is proposed.

To solve the problem of robot localization, each robot measures its peers' position and shares out the information. Each robot, armed with a point cloud of measurements of itself, computes a centroid to the points and feeds it to a Kalman filter. The Kalman filter tracks the state of the robot through a combination of predictions via kinematic equations and the measurements from the other robots.

The technique is tested in a simulation developed using Python scripting language. The performance of the data-sharing robot localization is compared with the performance of individual robot odometry, which is simulated in the simulation. From the simulation, it is observed that data-sharing robot localization should not be used when individual robot odometry errors are negligible, as the measurements themselves introduce errors. However, when the individual robot odometry errors are significant, data sharing can help to improve the performance. In fact, slower robots with fewer odometry errors can compensate for the odometry errors made by robots moving at a high speed.

B. RECOMMENDATIONS

The work of the thesis is a starting point in the exploration into collaborative robotics. In this thesis, the simulation makes several assumptions in view of the time constraint of the thesis work. However, further work can be done to address some of these assumptions, thereby producing a more accurate picture of the performance. The following are the recommendations for future work:

- Model the communication losses and delays between the robots.
- Model the line of sight of the robot when it is doing measurements of its peers.
- Include obstacles into the simulation to better reflect the reality of the urban environment.
- Model a wider range of sensors in the simulation to provide a view of what is necessary for effective robot localization by data sharing.

These recommendations are with respect to the assumptions of the simulation. With regard to the robot localization technique, further work can also be done to improve the capabilities. For example, the data-sharing capabilities of the robot can be expanded to include obstacle-sensing information for each robot. This can be built on to produce collaborative simultaneous localization and mapping (SLAM).

.

APPENDIX. SIMULATION SOURCE CODE

The simulation is developed in Python scripting language. The simulation is split into six files, botsim.py, bots.py, kf.py, simmanager.py, defines.py, and automation.py.

A. BOTSIM.PY

The botsim.py file contains the code to the main simulation code.

```
#!/usr/bin/python
import pygame
from pygame.locals import *
import os
import sys
from bots import BOTS
from defines import *
from simmanager import SimManager
import pickle

def runSim(num):
   x = 0
   y = 0
   os.environ['SDL_VIDEO_WINDOW_POS'] = "%d,%d" % (x,y)
   sim = SimManager()
   allSprites = pygame.sprite.RenderUpdates()
   clock = pygame.time.Clock()

   bot1 = BOTS((38,22,0),(0,0),(10,0), BOTSPEED, allSprites,'bot1', sim.screenSurface)
   allSprites.add(bot1)
   bot2 = BOTS((30,10,0),(180,0),(-10,0),BOTSPEED, allSprites,'bot2', 
sim.screenSurface)
   allSprites.add(bot2)
   bot3 = BOTS((45,20,0),(45,0),(-8,0),BOTSPEED, allSprites,'bot3', sim.screenSurface)
   allSprites.add(bot3)
   bot4 = BOTS((60,35,0),(135,0),(15,0),BOTSPEED, allSprites,'bot4', 
sim.screenSurface)
   allSprites.add(bot4)
   bot5 = BOTS((20,30,0),(90,0),(20,0), 1.2, allSprites,'bot5', sim.screenSurface)
   allSprites.add(bot5)
   bot6 = BOTS((35,40,0),(0,0),(-8,0), 1.2, allSprites,'bot6', sim.screenSurface)
   allSprites.add(bot6)

   """bot7 = BOTS((38,10,0),(0,0),(10,0), BOTSPEED, allSprites,'bot7', 
sim.screenSurface)
```

```
    allSprites.add(bot7)
    bot8 = BOTS((30,22,0),(0,0),(10,0), BOTSPEED, allSprites,’bot8’, sim.screenSurface)
    allSprites.add(bot8)
    bot9 = BOTS((45,22,0),(0,0),(10,0), BOTSPEED, allSprites,’bot9’, sim.screenSurface)
    allSprites.add(bot9)
    bot10    =    BOTS((60,10,0),(0,0),(10,0),    BOTSPEED,    allSprites,’bot10’,
sim.screenSurface)
    allSprites.add(bot10)
    bot11 = BOTS((20,35,0),(0,0),(-10,0), 1.2, allSprites,’bot11’, sim.screenSurface)
    allSprites.add(bot11)
    bot12 = BOTS((38,35,0),(0,0),(-10,0), 1.2, allSprites,’bot12’, sim.screenSurface)
    allSprites.add(bot12)”““

    if BEACONAVAIL:
        p = BOTS((10,10,0),(0,0),(0,0), 0, allSprites,’beacon’, sim.screenSurface)
        allSprites.add(p)




    displayBot = bot1
    displayBot.mark = 1

    while 1:
        #clock.tick(100)
        if bot1.datacount > NUMRUNS and NUMRUNS > 0:
            pygame.image.save(sim.windowScreen,.”/dat/run”+str(num)+.”png”)
            if PICKLINGMSE:
                file = open(.”/dat/data”+str(num)+.”pk,”“wb”)
                pickleDic = {}
                for sprite in allSprites:
                    if sprite.ID != “beacon”:
                        mse = {}
                        mse[“ODOMSE”] = sprite.msepdlist
                        mse[“SHAREMSE”] = sprite.mseppdlist
                        mse[“ODOPOS”] = [sprite.px,sprite.py,sprite.pz]
                        mse[“SHAREPOS”] = [sprite.mx,sprite.my,sprite.mz]
                        mse[“ACTUALPOS”] = [sprite.x,sprite.y,sprite.z]
                        pickleDic[sprite.ID] = mse
                pickle.dump(pickleDic, file)
                file.close()
                break
        else:
            if RENDERING:
                allSprites.clear(sim.botSurface, sim.screenSurface)
            for sprite in allSprites:
                sprite.updateBotMeas()
            for sprite in allSprites:
                sprite.updatePercep()
            if RUNKF:
                for sprite in allSprites:
                    sprite.updateKalman()
            allSprites.update()

            if RENDERING:
                sim.renderText(len(allSprites), displayBot)
                sim.render(allSprites)

        for event in pygame.event.get():
            if event.type == QUIT:
                sys.exit()
            elif event.type == KEYDOWN:
                if event.key == K_q:
                    pygame.image.save(sim.windowScreen,.”/dat/run.png”)

                    if PICKLINGMSE:
                        file = open(.”/dat/data.pk,”“wb”)
                        pickleDic = {}
                        for sprite in allSprites:
                            if sprite.ID != “beacon”:
                                mse = {}
                                mse[“ODOMSE”] = sprite.msepdlist
                                mse[“SHAREMSE”] = sprite.mseppdlist
                                mse[“ODOPOS”] = [sprite.px,sprite.py,sprite.pz]
                                mse[“SHAREPOS”] = [sprite.mx,sprite.my,sprite.mz]
                                mse[“ACTUALPOS”] = [sprite.x,sprite.y,sprite.z]
                                pickleDic[sprite.ID] = mse
                        pickle.dump(pickleDic, file)
                        file.close()

                    sys.exit()
                elif event.key == K_1:
                    displayBot.mark = 0
                    displayBot = bot1
                    displayBot.mark = 1
                elif event.key == K_2:
                    displayBot.mark = 0
                    displayBot = bot2
                    displayBot.mark = 1
                elif event.key == K_3:
                    displayBot.mark = 0
                    displayBot = bot3
                    displayBot.mark = 1
                elif event.key == K_4:
                    displayBot.mark = 0
                    displayBot = bot4
                    displayBot.mark = 1
                elif event.key == K_5:
                    displayBot.mark = 0
                    displayBot = bot5
                    displayBot.mark = 1
                elif event.key == K_6:
                    displayBot.mark = 0
                    displayBot = bot6
                    displayBot.mark = 1

if __name__ == '__main__':
    runSim(999)
```

B. BOTS.PY

The bots.py file contains the class that defines the robot behaviors.

```
import pygame
from defines import *
import numpy as np
from kf import KalmanFilter
import random

class BOTS(pygame.sprite.Sprite):
    def __init__(self,center,direction,steer,speed,spriteGrp,ID, screenSurface):
        pygame.sprite.Sprite.__init__(self)
        self.screenSurface = screenSurface
        self.image = pygame.Surface((30,30))
        self.image.fill((30,30,30,0))
        self.direction = direction[0]
        self.pdirection = direction[0]
        self.pitch = direction[1]
        self.ppitch = direction[1]
        self.r = 15
        self.image = self.image.convert_alpha()
        self.rect = self.image.get_rect()
        self.rect.center = (center[0],center[1])
        self.mark = 0
self.x = center[0]
self.y = center[1]
self.z = center[2]
self.px = self.x
self.py = self.y
self.pz = self.z
self.ppx = self.x
self.ppy = self.y
self.ppz = self.z
self.pmx = self.x
self.pmy = self.y
self.pmz = self.z
self.mx = self.x
self.my = self.y
self.mz = self.z
self.kfx = self.x
self.kfy = self.y
self.kfz = self.z
self.spriteGrp = spriteGrp
self.ID = ID
self.botList = {}
self.steer = steer[0]
self.pitchsteer = steer[1]
self.v = speed
self.msepx = 0
self.msepy = 0
self.msepz = 0
self.mseppx = 0
self.mseppy = 0
self.mseppz = 0
self.msepd = 0
self.mseppd = 0
self.msepdlist = []
self.mseppdlist = []
self.datacount = 0
self.drawBot()
t = TIMEDELTAPERSTEP
A = np.matrix([\
        [1,t,0,0,0,0],\
        [0,1,0,0,0,0],\
        [0,0,1,t,0,0],\
        [0,0,0,1,0,0],\
        [0,0,0,0,1,t],\
        [0,0,0,0,0,1]\
        ])
```

```
    H = np.eye(6)
    T = TIMEDELTAPERSTEP**2 / 2.0
    B = np.matrix([\
            [T,0,0],\
            [t,0,0],\
            [0,T,0],\
            [0,t,0],\
            [0,0,T],\
            [0,0,t]\
            ])
    Q = np.eye(6)*KFPROCESSERR
    R = np.eye(6)*KFMEASUREERR
    Vx = self.v * np.cos(np.deg2rad(self.direction)) * np.cos(np.deg2rad(self.pitch))
    Vy = self.v * np.sin(np.deg2rad(self.direction)) * np.cos(np.deg2rad(self.pitch))
    Vz = self.v * np.sin(np.deg2rad(self.pitch))
    xhat = np.matrix([\
            [self.x],\
            [Vx],\
            [self.y],\
            [Vy],\
            [self.z],\
            [Vz]\
            ])
    P = np.eye(6)
    self.filter = KalmanFilter(A,B,H,xhat,P,Q,R)


def getbotList(self):
    return self.botList
def getID(self):
    return self.ID
def getx(self):
    return self.x
def gety(self):
    return self.y
def getz(self):
    return self.z
def getpos(self):
    return np.matrix([self.x, self.y, self.z])
def getpx(self):
    return self.px
def getpy(self):
    return self.py
def getpz(self):
    return self.pz
```

```
def getppos(self):
    return np.matrix([self.px, self.py, self.pz])
def getpppos(self):
    return np.matrix([self.ppx, self.ppy, self.ppz])
def getdir(self):
    return self.direction
def getpitch(self):
    return self.pitch
def getpdir(self):
    return self.pdirection

def drawBot(self):
    # Determine points of triangle representing bot.
    frontPt = [15+int(self.r*np.cos(np.deg2rad(self.direction))),\
            15+int(self.r*np.sin(np.deg2rad(self.direction)))]
    leftPt = [15+int(self.r*np.cos(np.deg2rad(self.direction+150))),\
            15+int(self.r*np.sin(np.deg2rad(self.direction+150)))]
    rightPt = [15+int(self.r*np.cos(np.deg2rad(self.direction-150))),\
            15+int(self.r*np.sin(np.deg2rad(self.direction-150)))]
    self.image.fill((30,30,30,0))
    if self.mark:
        pygame.draw.polygon(self.image,COLORRED,[frontPt,leftPt,rightPt],0)
    else:
        pygame.draw.polygon(self.image,COLORBLUE,[frontPt,leftPt,rightPt],0)

def updateActualPos(self):
    # delta xy of actual position
    varianceX = ODOERR**2 * np.abs(self.v * \
            np.cos(np.deg2rad(self.direction)) * \
            np.cos(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP)
    varianceY = ODOERR**2 * np.abs(self.v * \
            np.sin(np.deg2rad(self.direction)) * \
            np.cos(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP)
    varianceZ = ODOERR**2 * np.abs(self.v * \
            np.sin(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP)

    dx = self.v * \
            np.cos(np.deg2rad(self.direction)) * \
            np.cos(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP + \
            random.gauss(0,np.sqrt(varianceX))
    dy = self.v * \
            np.sin(np.deg2rad(self.direction)) * \
            np.cos(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP + \
            random.gauss(0,np.sqrt(varianceY))

    dz = self.v * \
            np.sin(np.deg2rad(self.pitch)) * \
            TIMEDELTAPERSTEP + \
            random.gauss(0,np.sqrt(varianceZ))

    # update new actual position
    self.x += dx
    self.y += dy
    self.z += dz

def updatePerceivedPos(self):
    # delta xy of erroneous dead reckoning
    pdx = self.v * \
            np.cos(np.deg2rad(self.pdirection)) * \
            np.cos(np.deg2rad(self.ppitch)) * \
            TIMEDELTAPERSTEP
    pdy = self.v * \
            np.sin(np.deg2rad(self.pdirection)) * \
            np.cos(np.deg2rad(self.ppitch)) * \
            TIMEDELTAPERSTEP
    pdz = self.v * \
            np.sin(np.deg2rad(self.ppitch)) * \
            TIMEDELTAPERSTEP
    # update new perceived position
    self.px += pdx
    self.py += pdy
    self.pz += pdz
    # update data shared and kalman filtered perceived position with new perceived delta
    self.ppx = self.ppx + pdx
    self.ppy = self.ppy + pdy
    self.ppz = self.ppz + pdz

def updateDirections(self):
    # update new actual direction

    self.pdirection += self.steer * TIMEDELTAPERSTEP
    self.ppitch += self.pitchsteer * TIMEDELTAPERSTEP

    varianceDir = ODODIRERR**2 * np.abs(self.steer * TIMEDELTAPERSTEP)
```

```
        variancePitch     =     ODOPITCHERR**2     *     np.abs(self.pitchsteer     *
TIMEDELTAPERSTEP)

        # update new perceived erroneous direction
        self.direction = self.pdirection + \
            random.gauss(0,np.sqrt(varianceDir))

        self.pitch = self.ppitch + \
            random.gauss(0,np.sqrt(variancePitch))

    def renderPos(self):
        # update new position of Bot pixel position (pixels are integer only)
        # calculated x and y position are float for more precision
        dx = self.x - self.rect.centerx / PIXELSTOMETER
        dy = self.y - self.rect.centery / PIXELSTOMETER
        self.rect.move_ip(int(dx * PIXELSTOMETER), int(dy * PIXELSTOMETER))

    def renderActualPos(self):
            # Dot a pixel of actual path
        self.screenSurface.set_at((int(self.x     *     PIXELSTOMETER),     int(self.y     *
PIXELSTOMETER)),(100,100,100,255))

    def renderOdometryPos(self):
        # Dot a pixel of dead reckoning only perceived position
        self.screenSurface.set_at((int(self.px     *     PIXELSTOMETER),     int(self.py     *
PIXELSTOMETER)),(255,255,255,255))
    def renderDataSharePos(self):
        # Dot a pixel of data shared and Kalman filtered perceived position
        self.screenSurface.set_at((int(self.mx     *     PIXELSTOMETER),     int(self.my     *
PIXELSTOMETER)),(0,255,255,255))

    def renderCloud(self):
        if RENDERCLOUD and self.mark:
            for sprite in self.spriteGrp:
                if sprite != self:
                    if sprite.getbotList().has_key(self.ID):
                        x = sprite.getbotList()[self.ID][0]
                        y = sprite.getbotList()[self.ID][1]
                        self.screenSurface.set_at((int(x     *     PIXELSTOMETER),     int(y     *
PIXELSTOMETER)),COLORORANGE)


    def update(self):
        self.drawBot()
        self.updateDirections()
        self.updateActualPos()
        self.updatePerceivedPos()
        self.renderPos()
        self.renderActualPos()
        self.renderOdometryPos()
        self.renderDataSharePos()
        self.renderCloud()
        self.measMSE()

    def measMSE(self):
        self.msepx = (self.msepx * self.datacount +\
                (self.x - self.px)**2)/(self.datacount + 1)
        self.msepy = (self.msepy * self.datacount +\
                (self.y - self.py)**2)/(self.datacount + 1)
        self.msepz = (self.msepz * self.datacount +\
                (self.z - self.pz)**2)/(self.datacount + 1)
        self.msepd = (self.msepd * self.datacount +\
                np.linalg.norm(np.matrix([\
                        self.x - self.px, \
                        self.y - self.py, \
                        self.z - self.pz]))**2) / \
                (self.datacount + 1)

        self.mseppx = (self.mseppx * self.datacount +\
                (self.x - self.mx)**2)/(self.datacount + 1)
        self.mseppy = (self.mseppy * self.datacount +\
                (self.y - self.my)**2)/(self.datacount + 1)
        self.mseppz = (self.mseppz * self.datacount +\
                (self.z - self.mz)**2)/(self.datacount + 1)
        self.mseppd = (self.mseppd * self.datacount +\
                np.linalg.norm(np.matrix([self.x - self.ppx, \
                        self.y - self.ppy, \
                        self.z - self.ppz]))**2) / \
                (self.datacount + 1)

        self.datacount += 1
        if PICKLINGMSE:
            self.msepdlist.append(self.msepd)
            self.mseppdlist.append(self.mseppd)

    def updateBotMeas(self):
        “““calculate all the position estimated from data sharing”““
        self.botList = {}
        for sprite in self.spriteGrp:
            if sprite != self:
                # get vector between 2 bots
                diffmat = sprite.getpos() - self.getpos()

                # get actual distance between 2 bots
                dist = np.linalg.norm(diffmat)
                if dist < MEASUREMENTLIMIT:

                    # get actual bearing between 2 bots
                    bearing = np.rad2deg(np.arctan2(diffmat[0,1],diffmat[0,0]))
                    elevation = np.rad2deg(np.arcsin(diffmat[0,2] / np.linalg.norm(diffmat)))

                    # add Gaussian error to bearing between 2 bots
                    if VARMEASUREERR:
                        accuracyFound = 0
                        accuIter = iter(BEARINGERR)
                        while not accuracyFound:
                            spec = accuIter.next()
                            if dist < spec[0]:
                                bearingAccu = spec[1]
                                elevationAccu = spec[2]
                                accuracyFound = 1
                        bearing += random.uniform(-bearingAccu,bearingAccu)
                        elevation += random.uniform(-elevationAccu,elevationAccu)
                    else:
                        bearing += random.uniform(-FIXBEARINGERR,FIXBEARINGERR)
                        elevation                    +=                    random.uniform(-
FIXELEVATIONERR,FIXELEVATIONERR)

                    # add Gaussian error to distance between 2 bots
                    if VARMEASUREERR:
                        accuracyFound = 0
                        accuIter = iter(DISTERR)
                        while not accuracyFound:
                            spec = accuIter.next()
                            if dist < spec[0]:
                                accu = spec[1]
                                accuracyFound = 1
                        dist += random.uniform(-accu,accu)
                    else:
                        dist += random.uniform(-FIXDISTERR,FIXDISTERR)

                    # calculate estimated position of observed bot based on
                    # data shared and kalman filtered perceived position of observer
                    # using erroneous distance measurement and erroneous bearing measurement
```

```
            x = self.ppx + dist*np.cos(np.deg2rad(bearing))
            y = self.ppy + dist*np.sin(np.deg2rad(bearing))
            z = self.ppz + dist*np.sin(np.deg2rad(elevation))

            # observer maintain a list of estimated positions of observed bots
            self.botList[sprite.getID()] = [x, y, z, dist]


def updatePercep(self):
    “““calculate better estimate of self position by obtaining
    list of position estimate of self from all the other observer
    bots the centroid of the point cloud is calculated and used
    as new estimate of self position”““
    if self.v == 0:
        pass
    else:
        sumx = 0
        sumy = 0
        sumz = 0
        count = 0
        maxDist = 0
        for sprite in self.spriteGrp:
            if sprite != self:
                if sprite.getbotList().has_key(self.ID):
                    if sprite.getbotList()[self.ID][3] > maxDist:
                        maxDist = sprite.getbotList()[self.ID][3]
        for sprite in self.spriteGrp:
            if sprite != self:
                # if observer bot has an estimate of self in
                # it’s estimate list add point to point cloud
                if sprite.getbotList().has_key(self.ID):
                    x = sprite.getbotList()[self.ID][0]
                    y = sprite.getbotList()[self.ID][1]
                    z = sprite.getbotList()[self.ID][2]
                    dist = sprite.getbotList()[self.ID][3]
                    if WEIGHTEDCENTROID:
                        sumx += x * (maxDist / dist**WEIGHTINTENSITY)
                        sumy += y * (maxDist / dist**WEIGHTINTENSITY)
                        sumz += z * (maxDist / dist**WEIGHTINTENSITY)
                        count += 1.0 * (maxDist / dist**WEIGHTINTENSITY)
                    else:
                        sumx += x
                        sumy += y
                        sumz += z
                        count += 1.0
        # store previous measured position
        self.pmx = self.mx
        self.pmy = self.my
        self.pmz = self.mz

        if count > 0:
            self.mx = sumx / count * 1.0
            self.my = sumy / count * 1.0
            self.mz = sumz / count * 1.0
        else:
            self.mx = self.ppx
            self.my = self.ppy
            self.mz = self.ppz

def updateKalman(self):
    if self.v > 0:
        radD = np.deg2rad(self.pdirection)
        radSteer = np.deg2rad(self.steer * TIMEDELTAPERSTEP)
        radPitch =  np.deg2rad(self.ppitch)
        radPitchSteer = np.deg2rad(self.pitchsteer * TIMEDELTAPERSTEP)
        currVectx = self.v * \
                np.cos(radD + radSteer) * \
                np.cos(radPitch + radPitchSteer)
        prevVectx = self.v * \
                np.cos(radD) * \
                np.cos(radPitch)
        currVecty = self.v * \
                np.sin(radD + radSteer) * \
                np.cos(radPitch + radPitchSteer)
        prevVecty = self.v * \
                np.sin(radD) * \
                np.cos(radPitch)
        currVectz = self.v * \
                np.sin(radPitch + radPitchSteer)
        prevVectz = self.v * \
                np.sin(radPitch)
        accX = (currVectx - prevVectx) / TIMEDELTAPERSTEP
        accY = (currVecty - prevVecty) / TIMEDELTAPERSTEP
        accZ = (currVectz - prevVectz) / TIMEDELTAPERSTEP
        u = np.matrix([\
                [accX],\
                [accY],\
                [accZ]\
                ])
        m = np.matrix([\
            [self.mx],\
            [(self.mx - self.pmx) / TIMEDELTAPERSTEP],\
            [self.my],\
            [(self.my - self.pmy) / TIMEDELTAPERSTEP],\
            [self.mz],\
            [(self.mz - self.pmz) / TIMEDELTAPERSTEP]\
            ])

        self.filter.Step(u,m)

        currState = self.filter.GetCurrState()
        self.kfx = currState[0,0]
        self.kfy = currState[2,0]
        self.kfz = currState[4,0]
        self.ppx = self.kfx
        self.ppy = self.kfy
        self.ppz = self.kfz
        self.mx = self.kfx
        self.my = self.kfy
        self.mz = self.kfz
```

C. KF.PY

The kf.py file contains the code to implement the Kalman filter.

```
import numpy as np

class KalmanFilter:
  def __init__(self,A, B, H, x, P, Q, R):
    self.A = A
    self.B = B
    self.H = H
    self.currState = x
    self.currProb = P
    self.Q = Q
    self.R = R
  def GetCurrState(self):
    return self.currState
  def Step(self,u,m):
    self.u = u
    predState = self.A * self.currState + self.B * self.u
    self.predState = predState
    predProb = (self.A * self.currProb) * np.transpose(self.A) + self.Q
    y = m - self.H * predState
```

```
        S = self.H * predProb * np.transpose(self.H) + self.R
        K = predProb * np.transpose(self.H) * np.linalg.inv(S)
        self.currState = predState + K * y
        size = self.currProb.shape[0]
        self.currProb = (np.eye(size)- K * self.H) * predProb
```

D. SIMMANAGER.PY

The simmanager.py file contains the graphics handler for the simulation.

```
import pygame
from pygame.locals import *
from defines import *

class SimManager:

    def __init__(self):
        pygame.init()
        if RENDERING:
            self.windowScreen = pygame.display.set_mode((1600, 900), FULLSCREEN)
            pygame.display.set_caption('Orbits!')
            self.screenSurface = pygame.Surface((1400,900))
            self.screenSurface = self.windowScreen.convert()
            self.screenSurface.fill((30, 30, 30))
            self.botSurface = pygame.Surface((1400,900))
            self.menuSurface = pygame.Surface((200,900))
            self.menuSurface.fill(COLORGREEN)
            self.windowScreen.blit(self.screenSurface,(200,0))
            self.windowScreen.blit(self.menuSurface, (0,0))
            pygame.display.flip()
            self.fontObj = pygame.font.Font('freesansbold.ttf',18)
        else:
            self.windowScreen = pygame.display.set_mode((1, 1))
            self.screenSurface = pygame.Surface((1400,900))

    def render(self, allSprites):
        self.botSurface.blit(self.screenSurface,(0,0))
        allSprites.draw(self.botSurface)
        self.windowScreen.blit(self.botSurface,(200,0))
        self.windowScreen.blit(self.menuSurface,(0,0))

        pygame.draw.line(self.windowScreen,         COLORWHITE,         (300,10),
(300+10*PIXELSTOMETER,10), 5)
        pygame.display.flip()
    def renderText(self, numSprites, bot):
        self.menuSurface.fill(COLORGREEN, pygame.Rect(0,0,200,380))
        textline = 20
        self.menuSurface.blit(self.fontObj.render("Number of bots," True, COLORBLACK),
(10,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render(str(numSprites), True, COLORBLACK),
(15,textline))
        textline += 40
        self.menuSurface.blit(self.fontObj.render(bot.ID +" Data," True, COLORBLACK),
(10,textline))
        textline += 30
        self.menuSurface.blit(self.fontObj.render("Robot speed m/s:," True,
COLORBLACK), (10,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render(str(bot.v), True, COLORBLACK),
(15,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render("Robot steer deg/s," True,
COLORBLACK), (10,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render(str(bot.steer), True, COLORBLACK),
(15,textline))

        textline += 30
        self.menuSurface.blit(self.fontObj.render("Odometry:," True, COLORBLACK),
(10,textline))
        textline += 30
        # self.menuSurface.blit(self.fontObj.render("Mean Sq Err X:," True,
COLORBLACK), (10,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render(str(bot.msepx), True, COLORBLACK),
(15,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render("Mean Sq Err Y:," True,
COLORBLACK), (10,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render(str(bot.msepy), True, COLORBLACK),
(15,textline))
        # textline += 20
        self.menuSurface.blit(self.fontObj.render("Mean Sq Err dist:," True,
COLORBLACK), (10,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render(str(bot.msepd), True, COLORBLACK),
(15,textline))
```

```
        textline += 40
        self.menuSurface.blit(self.fontObj.render("Data sharing:," True, COLORBLACK),
(10,textline))
        textline += 30
        # self.menuSurface.blit(self.fontObj.render("Mean Sq Err X:," True,
COLORBLACK), (10,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render(str(bot.mseppx), True, COLORBLACK),
(15,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render("Mean Sq Err Y:," True,
COLORBLACK), (10,textline))
        # textline += 20
        # self.menuSurface.blit(self.fontObj.render(str(bot.mseppy), True, COLORBLACK),
(15,textline))
        # textline += 20
        self.menuSurface.blit(self.fontObj.render("Mean Sq Err dist:," True,
COLORBLACK), (10,textline))
        textline += 20
        self.menuSurface.blit(self.fontObj.render(str(bot.mseppd), True, COLORBLACK),
(15,textline))
```

E. DEFINES.PY

The defines.py file contains the input parameters to the simulation as well as the options for configuring the simulation.

```
import pygame

RENDERING = 0

FLYING = 0

BEACONAVAIL = 0
NUMRUNS = 1000

MEASUREMENTLIMIT = 30.0

BOTSPEED = 0.6
ODODIRERR = 0.1
ODOERR = 0.5
if FLYING:
   ODOPITCHERR = 0.1
else:
   ODOPITCHERR = 0.0
```

```
PIXELSTOMETER = 20.0
TIMEDELTAPERSTEP = 0.1

WEIGHTEDCENTROID = 0
WEIGHTINTENSITY = 2.0

VARMEASUREERR = 1
if FLYING:
   BEARINGERR = [[10,0.25,0.25],[MEASUREMENTLIMIT,0.25,0.25]]
   FIXELEVATIONERR = 0.25
else:
   BEARINGERR = [[10,0.25,0],[MEASUREMENTLIMIT,0.25,0]]
   FIXELEVATIONERR = 0.0
DISTERR = [[10,0.01],[MEASUREMENTLIMIT,0.03]]

FIXDISTERR = 0.03
FIXBEARINGERR = 0.25

RUNKF = 1
KFPROCESSERR = 0.5
KFMEASUREERR = 1.0

RENDERCLOUD = 0

PICKLINGMSE = 1

COLORRED = pygame.Color(255,0,0)
COLORORANGE = pygame.Color(255,180,0)
COLORGREEN = pygame.Color(0,255,0)
COLORPEPPERMINT = pygame.Color(0,255,100)
COLORBLUE = pygame.Color(0,0,255)
COLORWHITE = pygame.Color(255,255,255)
COLORBLACK = pygame.Color(0,0,0)
```

F. AUTOMATE.PY

The automate.py file is a helper file to assist in running multiple simulation runs automatically.

```
import botsim
import time

for i in range(1000):
   start = time.time()
   botsim.runSim(i)
```

```
print “Run: “+str(i)+,” Dur: “+str(time.time()-start)
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Greg Welch and Gary Bishop. 2006. *An Introduction to the Kalman Filter.* TR 95-041.Chapel Hill: University of North Carolina at Chapel Hill.

Hui Fang, Wen-Jing Hsu, and Larry Rudolph. 2008. "Controlling Uncertainty in Personal Positioning at Minimal Measurement Cost." In *Ubiquitous Intelligence and Computing*, 468–481. Berlin Heidelberg: Springer.

Ingemar J. Cox. 1990. "Blanche: Position Estimation for an Autonomous Robot Vehicle." In *Autonomous Robot Vehicles,* edited by Ingemar J. Cox and Gordon T. Wilfong, 221–228. New York: Springer.

Ivanka Terzic, Alois Zoitl, Bernard Favre, and Thomas Strasser. 2008. "A Survey of Distributed Intelligence in Automation in European Industry, Research and Market." In *Institute of Electrical and Electronics Engineers (IEEE) International Conference on Emerging Technologies and Factory Automation*, 221–228. Piscataway: Institute of Electrical and Electronics Engineers (IEEE).

Lei Zhang, Rene Zapata, and Pascal Lepinay. 2009. "Self-Adaptive Monte Carlo for Single-Robot and Multi-Robot Localization." In *Institute of Electrical and Electronics Engineers (IEEE) International Conference on Automation and Logistics*, 1927–1933. Piscataway: Institute of Electrical and Electronics Engineers (IEEE).

Lynne E. Parker. 2008. "Distributed Intelligence: Overview of the Field and its Application in Multi-Robot Systems." *Journal of Physical Agents* 2(1): 5–14.

Mohinder S. Grewal and Angus P. Andrews. 2014. *Kalman Filtering: Theory and Practice with MATLAB*. 4th ed. Hoboken: John Wiley & Sons.

Rudy Negenborn. 2003. "Robot Localization and Kalman Filters." PhD diss., Utrecht University.

Sebastian Thrun. Wolfram Burgard, and Dieter Fox. 2005. *Probabilistic Robotics*. Cambridge: MIT Press.

Sebastian, Thrun. 2002. "Robotic Mapping: A Survey." In *Exploring Artificial Intelligence in the New Millennium,* edited by Gerhard Lakemeyer and Bernhard Nebel, 1–35. San Francisco: Morgan Kaufmann Publishers Inc.

Tomas Krajnik, Matias Nitsche, Jan Faigl, Tom Duckett, Marta Mejail, and Libor Preucil. 2013. "External Localization System for Mobile Robotics." In *International Conference on Advanced Robotics* (ICAR), 1–6. Piscataway: IEEE.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California